SIMULINK®

Dynamic System Simulation for MATLAB®

Modeling

Simulation

Implementation

User's Guide

Version 2.1

How to Contact The MathWorks:

508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 24 Prime Park Way Natick, MA 01760-1500	Mail
`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

Simulink User's Guide

Printing History:	March 1992	First printing	
	May 1997	Revised for 2.1	(online version)

Reprints - December 1993, November 1994, June 1995, July 1996

Contents

1 Getting Started

2 Quick Start

3 Creating a Model

4 Running a Simulation

5 Analyzing Simulation Results

6 Using Masks to Customize Blocks

7 Conditionally Executed Subsystems

8 S-Functions

9 Block Reference

10 Additional Topics

11 Model Construction Commands

A Model and Block Parameters

B Model File Format

C The SimStruct

1

Getting Started

To the Reader

Welcome to Simulink®! In the last few years, Simulink has become the most widely used software package in academia and industry for modeling and simulating dynamical systems.

Simulink encourages you to try things out. You can easily build models from scratch, or take an existing model and add to it. Simulations are interactive, so you can change parameters "on the fly" and immediately see what happens. You have instant access to all of the analysis tools in MATLAB®, so you can take the results and analyze and visualize them. We hope that you will get a sense of the *fun* of modeling and simulation, through an environment that encourages you to pose a question, model it, and see what happens.

With Simulink, you can move beyond idealized linear models to explore more realistic nonlinear models, factoring in friction, air resistance, gear slippage, hard stops, and the other things that describe real-world phenomena. It turns your computer into a lab for modeling and analyzing systems that simply wouldn't be possible or practical otherwise, whether the behavior of an automotive clutch system, the flutter of an airplane wing, the dynamics of a predator-prey model, or the effect of the monetary supply on the economy.

Simulink is also practical. With thousands of engineers around the world using it to model and solve real problems, knowledge of this tool will serve you well throughout your professional career.

We hope you enjoy exploring the software.

What Is Simulink?

Simulink is a software package for modeling, simulating, and analyzing dynamical systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can be also multirate, i.e., have different parts that are sampled or updated at different rates.

For modeling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. With this interface, you can draw the models just as you would with pencil and paper (or as most textbooks depict them). This is a far cry from previous simulation packages that require you to formulate differential equations and difference equations in a language or program. Simulink includes a comprehensive block

library of sinks, sources, linear and nonlinear components, and connectors. You can also customize and create your own blocks.

Models are hierarchical, so you can build models using both top-down and bottom-up approaches. You can view the system at a high-level, then double-click on blocks to go down through the levels to see increasing levels of model detail. This approach provides insight into how a model is organized and how its parts interact.

After you define a model, you can simulate it, using a choice of integration methods, either from the Simulink menus or by entering commands in MATLAB's command window. The menus are particularly convenient for interactive work, while the command-line approach is very useful for running a batch of simulations (for example, if you are doing Monte Carlo simulations or want to sweep a parameter across a range of values). Using scopes and other display blocks, you can see the simulation results while the simulation is running. In addition, you can change parameters and immediately see what happens, for "what if" exploration. The simulation results can be put in the MATLAB workspace for postprocessing and visualization.

Model analysis tools include linearization and trimming tools, which can be accessed from the MATLAB command line, plus the many tools in MATLAB and its application toolboxes. And because MATLAB and Simulink are integrated, you can simulate, analyze, and revise your models in either environment at any point.

How to Use This Manual

Because Simulink is graphical and interactive, we encourage you to jump right in and try it.

The manual contains eleven chapters and five appendices. For a useful introduction that will help you start using Simulink quickly, take a look at "Running a Demo Model" in Chapter 2. Browse around the model, double-click on blocks that look interesting, and you will quickly get a sense of how Simulink works. If you want a quick lesson in building a model, see "Building a Simple Model" in Chapter 2.

Chapter 3 describes in detail how to build and edit a model. It also discusses how to save and print a model and provides some useful tips.

Chapter 4 describes how Simulink performs a simulation. It covers simulation parameters and the integration solvers used for simulation, including some of

the strengths and weaknesses of each solver that should help you choose the appropriate solver for your problem. It also discusses multirate and hybrid systems.

Chapter 5 discusses Simulink and MATLAB features useful for viewing and analyzing simulation results.

Chapter 6 discusses methods for creating your own blocks and using masks to customize their appearance and use.

Chapter 7 describes subsystems whose execution depends on triggering signals.

Chapter 8 describes how to create blocks using M-files or C MEX-files.

Chapter 9 provides reference information for all Simulink blocks.

Chapter 10 provides information about how Simulink works, including information about zero crossings, algebraic loops, and discrete and hybrid systems.

Chapter 11 provides reference information for commands you can use to create and modify a model from the MATLAB command window or from an M-file.

Appendix A lists model and block parameters. This information is useful with the `get_param` and `set_param` commands, described in Chapter 11.

Appendix B describes the format of the file that stores model information.

Appendix C provides the contents of the SimStruct, the data structure that describes S-functions.

Although we have tried to provide the most complete and up-to-date information in this manual, some information may have changed after it was printed. Please check the *Simulink Late-Breaking News*, delivered with your Simulink system, for the latest release notes.

Professional Application Toolboxes

One of the key features of Simulink is that it is built atop MATLAB. As a result, Simulink users have direct access to the wide range of MATLAB-based tools for generating, analyzing, and optimizing systems implemented in Simulink. These tools include MATLAB Application Toolboxes, specialized collections of M-files for working on particular classes of problems.

Toolboxes are more than just collections of useful functions; they represent the efforts of some of the world's top researchers in fields such as controls, signal processing, and system identification. MATLAB Application Toolboxes therefore let you "stand on the shoulders" of world class scientists.

All toolboxes are built using MATLAB. This has some very important implications for you:

- Every toolbox builds on the robust numerics, rock-solid accuracy, and years of experience in MATLAB.
- You get seamless and immediate integration with Simulink and any other toolboxes you may own.
- Because all toolboxes are written in MATLAB code, you can take advantage of MATLAB's open-system approach. You can inspect M-files, add to them, or use them for templates when creating your own functions.
- Every toolbox is available on any computer platform that runs MATLAB.

Here is a list of professional toolboxes currently available from The MathWorks. This list is by no means static— more are being created every year.

The Communications Toolbox. The Communications Toolbox provides an integrated set of tools for accelerating the design, analysis, and simulation of modern communications systems. It combines MATLAB's high-level language with the ease of use of Simulink's block diagram interface, and provides communications engineers with comprehensive communications system design and analysis capabilities. The toolbox is useful in such diverse industries as telecommunications, telephony, aerospace, and computer peripherals.

The Control System Toolbox. The Control System Toolbox, the foundation of the MATLAB control design toolbox family, contains functions for modeling, analyzing, and designing automatic control systems. The application of automatic control grows each year as sensors and computers become less expensive. As a result, automatic controllers are used not only in highly technical settings for automotive and aerospace systems, computer peripherals, and process control, but also in less obvious applications such as washing machines and cameras.

The Financial Toolbox. The Financial Toolbox operates with MATLAB to provide a robust set of financial functions essential to financial and quantitative analysis. Applications include pricing securities, calculating interest and yield, analyzing derivatives, and optimizing portfolios. The Financial Toolbox requires the Statistics and Optimization Toolboxes. The Simulink graphical interface is recommended for Monte Carlo and non-stochastic simulations for pricing fixed-income securities, derivatives, and other instruments.

The Financial Toolbox includes functions for the input, processing, and output of financial data:

- Fixed-income pricing, yield, and sensitivity routines
- Cash flow evaluation and financial accounting functions
- Derivatives analysis procedures
- Portfolio analysis tools
- Date functions
- Graphic formats and cash formatting functions

The Frequency-Domain System Identification Toolbox. The Frequency-Domain System Identification Toolbox by István Kollár, in cooperation with Johan Schoukens and researchers at the Vrije Universiteit in Brussels, is a set of M-files for modeling linear systems based on measurements of the system's frequency response.

The Fuzzy Logic Toolbox. The Fuzzy Logic Toolbox provides a complete set of GUI-based tools for designing, simulating, and analyzing fuzzy inference systems. Fuzzy logic provides an easily understandable, yet powerful way to map an input space to an output space with arbitrary complexity, with rules and relationships specified in natural language. Systems can be simulated in

MATLAB or incorporated into a Simulink block diagram, with the ability to generate code for stand-alone execution.

The Higher-Order Spectral Analysis Toolbox. The Higher-Order Spectral Analysis Toolbox, by Jerry Mendel, C. L. (Max) Nikias, and Ananthram Swami, provides tools for signal processing using higher-order spectra. These methods are particularly useful for analyzing signals originating from a nonlinear process or corrupted by non-Gaussian noise.

The Image Processing Toolbox. The Image Processing Toolbox contains tools for image processing and algorithm development. It includes tools for filter design and image restoration; image enhancement; analysis and statistics; color, geometric, and morphological operations; and 2-D transforms.

The LMI Control Toolbox. The LMI Control Toolbox, authored by leading researchers: Pascal Gahinet, Arkadi Nemirovski, and Alan Laub, allows one to efficiently solve Linear Matrix Inequalities (LMIs). LMIs are special convex optimization problems that arise in many disciplines, including control, identification, filtering, structural design, graph theory, and linear algebra.

The LMI Control Toolbox also features a variety of LMI-based tools for control systems design and covers applications such as robust stability and performance analysis, robust gain scheduling, and multi-objective controller synthesis with a mix of H-infinity, LQG, and pole placement objectives.

The Model Predictive Control Toolbox. The Model Predictive Control Toolbox was written by Manfred Morari and N. Lawrence Ricker. Model predictive control is especially useful for control applications with many input and output variables, many of which have constraints. As a result, it has become particularly popular in chemical engineering and other process control applications.

The Mu-Analysis and Synthesis Toolbox. The Mu-Analysis and Synthesis Toolbox, by Gary Balas, Andy Packard, John Doyle, Keith Glover, and Roy Smith, contains specialized tools for H_∞ optimal control, and μ-analysis and synthesis, an approach to advanced robust control design of multivariable linear systems.

The NAG Foundation Toolbox. The NAG Foundation Toolbox includes more than 200 numeric computation functions from the well-regarded NAG Fortran subroutine libraries. It provides specialized tools for boundary-value problems,

optimization, adaptive quadrature, surface and curve-fitting, and other applications.

The Neural Network Toolbox. The Neural Network Toolbox by Howard Demuth and Mark Beale is a collection of MATLAB functions for designing and simulating neural networks. Neural networks are computing architectures, inspired by biological nervous systems, that are useful in applications where formal analysis is extremely difficult or impossible, such as pattern recognition and nonlinear system identification and control.

The Optimization Toolbox. The Optimization Toolbox contains commands for the optimization of general linear and nonlinear functions, including those with constraints. An optimization problem can be visualized as trying to find the lowest (or highest) point in a complex, highly contoured landscape. An optimization algorithm can thus be likened to an explorer wandering through valleys and across plains in search of the topographical extremes.

The Partial Differential Equation Toolbox. The Partial Differential Equation Toolbox extends the MATLAB Technical Computing Environment for the study and solution of PDEs in two space dimensions (2-D) and time. The PDE Toolbox provides a set of command line functions and an intuitive graphical user interface for preprocessing, solving, and postprocessing generic 2-D PDEs using the Finite Element Method (FEM). The toolbox also provides automatic and adaptive meshing capabilities and a suite of eight application modes for common PDE application areas such as heat transfer, structural mechanics, electrostatics, magnetostatics, and diffusion. These application areas are common in the fields of engineering and physics.

The QFT Control Design Toolbox. The Quantitative Feedback Theory Toolbox by Yossi Chait, Craig Borghesani, and Oded Yaniv implements QFT, a frequency-domain approach to controller design for uncertain systems that provides direct insight into the trade-offs between controller complexity (hence the ability to implement it) and specifications.

The Robust Control Toolbox. The Robust Control Toolbox provides a specialized set of tools for the analysis and synthesis of control systems that are “robust” with respect to uncertainties that can arise in the real world. The Robust Control Toolbox was created by controls theorists Richard Y. Chiang and Michael G. Safonov.

The Signal Processing Toolbox. The Signal Processing Toolbox contains tools for signal processing. Applications include audio (e.g., compact disc and digital audio tape), video (digital HDTV, image processing, and compression), telecommunications (fax and voice telephone), medicine (CAT scan, magnetic resonance imaging), geophysics, and econometrics.

The Spline Toolbox. The Spline Toolbox by Carl de Boor, a pioneer in the field of splines, provides a set of M-files for constructing and using splines, which are piecewise polynomial approximations. Splines are useful because they can approximate other functions without the unwelcome side effects that result from other kinds of approximations, such as piecewise linear curves.

The Statistics Toolbox. The Statistics Toolbox provides a set of M-files for statistical data analysis, modeling, and Monte Carlo simulation, with GUI-based tools for exploring fundamental concepts in statistics and probability.

The Symbolic Math Toolbox. The Symbolic Math Toolbox gives MATLAB an integrated set of tools for symbolic computation and variable-precision arithmetic, based on Maple V®. The Extended Symbolic Math Toolbox adds support for Maple programming plus additional specialized functions.

The System Identification Toolbox. The System Identification Toolbox, written by Lennart Ljung, is a collection of tools for estimation and identification. System identification is a way to find a mathematical model for a physical system (like an electric motor, or even a financial market) based only on a record of the system's inputs and outputs.

The Wavelet Toolbox. The Wavelet Toolbox provides a comprehensive collection of routines for examining local, multiscale, or nonstationary phenomena. Wavelet methods offer additional insight and performance in any application where Fourier techniques have been used. The toolbox is useful in many signal and image processing applications, including speech and audio processing, communications, geophysics, finance, and medicine.

The Simulink Real-Time Workshop

The Simulink Real-Time Workshop™ automatically generates C code directly from Simulink block diagrams. This allows the execution of continuous, discrete-time, and hybrid system models on a wide range of computer platforms, including real-time hardware. Simulink is required.

The Real-Time Workshop can be used for:

- Rapid Prototyping. As a rapid prototyping tool, the Real-Time Workshop enables you to implement your designs quickly without lengthy hand coding and debugging. Control, signal processing, and dynamic system algorithms can be implemented by developing graphical Simulink block diagrams and automatically generating C code.
- Embedded Real-Time Control. Once a system has been designed with Simulink, code for real-time controllers or digital signal processors can be generated, cross-compiled, linked, and downloaded onto your selected target processor. The Real-Time Workshop supports DSP boards, embedded controllers, and a wide variety of custom and commercially available hardware.
- Real-Time Simulation. You can create and execute code for an entire system or specified subsystems for hardware-in-the-loop simulations. Typical applications include training simulators (pilot-in-the-loop), real-time model validation, and testing.
- Stand-Alone Simulation. Stand-alone simulations can be run directly on your host machine or transferred to other systems for remote execution. Because time histories are saved in MATLAB as binary or ASCII files, they can be easily loaded into MATLAB for additional analysis or graphic display.

Key Features

Real-Time Workshop provides a comprehensive set of features and capabilities that provide the flexibility to address a broad range of applications.

- Automatic code generation handles continuous-time, discrete-time, and hybrid systems.
- Optimized code guarantees fast execution.
- Control framework API uses customizable makefiles to build and download object files to target hardware automatically.

- Portable code facilitates usage in a wide variety of environments.
- Concise, readable, and well-documented code provides ease of maintenance.
- Interactive parameter downloading from Simulink to external hardware allows system tuning on the fly.
- A menu-driven, graphical user interface makes the software easy to use.

The Real-Time Workshop supports the following target environments:

- dSPACE DS1102, DS1002, DS1003 using TI C30/C31/C40 DSPs
- VxWorks, VME/68040
- 486 PC-based systems with Xycom, Matrix, Data Translation, or Computer Boards I/O devices and Quanser Multiq board

The Real-Time Workshop Ada Extension

The Simulink Real-Time Workshop (RTW) Ada Extension automatically generates Ada code directly from Simulink block diagrams. This allows the execution of continuous, discrete-time, and hybrid system models on a wide range of computer platforms, including real-time hardware. Simulink is required.

RTW Ada Extension can be used for:

- Rapid Prototyping. As a rapid prototyping tool, the RTW Ada Extension enables you to implement your designs quickly without lengthy hand coding and debugging. Control and dynamic system algorithms can be implemented by developing graphical Simulink block diagrams and automatically generating Ada code.
- Embedded Real-Time Control. Once a system has been designed with Simulink, code for real-time controllers can be generated, cross-compiled, linked, and downloaded onto your selected target processor. The RTW Ada Extension generates Ada code, which can be run on a wide variety of custom and commercially available hardware.
- Real-Time Simulation. You can create and execute code for an entire system or specified subsystems for hardware-in-the-loop simulations. Typical applications include training simulators (pilot-in-the-loop), real-time model validation, and testing.
- Stand-Alone Simulation. Stand-alone simulations can be run directly on your host machine or transferred to other systems for remote execution. Because time histories are saved in MATLAB as binary or ASCII files, they can be easily loaded into MATLAB for additional analysis or graphic display.

Key Features

RTW Ada Extension provides a comprehensive set of features and capabilities that provide the flexibility to address a broad range of applications.

- Automatic code generation handles continuous-time, discrete-time, and hybrid systems.
- Optimized code guarantees fast execution.
- Control framework API uses customizable makefiles to build and download object files to target hardware automatically.

- Portable code facilitates usage in a wide variety of environments.
- Concise, readable, and well-commented code provides ease of maintenance.
- A menu-driven, graphical user interface makes it easy to use.

The RTW Ada Extension provides turnkey solutions for the following Ada 83 compilers:

- Rational VADS for UNIX platforms
- Thomson ActivAda for Windows Professional Edition
- Thomson ActivAda for Windows NT

Blocksets

Similar to MATLAB and its application toolboxes, The MathWorks offers Blocksets for use with Simulink. Blocksets are collections of Simulink blocks that are grouped in a separate library from the main Simulink library.

The DSP Blockset

The DSP Blockset extends Simulink for use in the rapid design and simulation of DSP-based devices and systems. With the DSP Blockset, Simulink provides an intuitive tool for interactive block-diagram simulation and evaluation of signal processing algorithms. Its graphical programming environment makes it easier for engineers to create, modify, and prototype DSP designs. Simulink is required.

Applications for the DSP Blockset include design and analysis of communications systems, computer peripherals, speech and audio processing, automotive and aerospace controls, and medical electronics. It is ideal for both time and frequency domain algorithms, including problems such as adaptive noise cancellation.

The Fixed-Point Blockset

The Fixed-Point Blockset, for use with Simulink, includes a collection of block diagram components that extend the standard Simulink block library. With this new set of blocks, you can create discrete-time dynamic systems that utilize fixed-point arithmetic. As a result, Simulink can simulate effects commonly encountered in fixed-point systems for applications such as control systems and time-domain filtering. Simulink is required.

The Fixed-Point Blockset allows you to simulate fixed-point effects in a convenient and productive environment. The new blocks provided by the Fixed-Point Blockset include blocks for:

- Addition and subtraction
- Multiplication and division
- Summation
- Gains and constants
- Conversion between floating-point and fixed-point signals
- 1- and 2-D lookup tables

- Logical operators
- Relational operators
- Conversion/saturation of fixed-point signals
- Switch between two values
- Delay
- Delta-inverse operator
- Monitoring signals

Signal conversion blocks let you convert between floating-point and fixed-point signals. Using the conversion blocks, you can create Simulink block diagrams, which consist of both standard Simulink block library components and fixed-point blocks.

For example, you can create plant models using the standard Simulink blocks and model the controller with fixed-point blocks. Data range blocks provide maximum and minimum values encountered during simulation from any point in the block diagram.

The Fixed-Point Blockset lets you build models using unsigned or 2's complement 8-, 16-, or 32-bit word lengths. A combination of blocks with differing word lengths may be used in the same block diagram. Scaling of fixed-point values is achieved by specifying the location of the binary-point within the fixed-point blocks. During simulation, data types can be changed allowing you to immediately see the effects of different word sizes, binary-point locations, rounding versus truncation, and overflow checking.

Another powerful feature of this blockset is automatic location of the binary-point to give maximum precision without overflow.

By using the data range blocks, you can fix binary point locations to appropriate values.

The Fixed-Point Blockset requires Simulink 1.3 and MATLAB 4.2c and is currently shipping on PC and Macintosh.

The Nonlinear Control Design Blockset

The Nonlinear Control Design (NCD) Blockset offers time domain-based, robust, nonlinear control design. Controller designs are developed as block diagrams in Simulink. You select a set of tunable model parameters and

graphically place time response constraints on selected output signals. Successive simulation and optimization methods are applied automatically, thereby tuning the selected model parameters.

Simulink is required with the NCD Blockset.

2

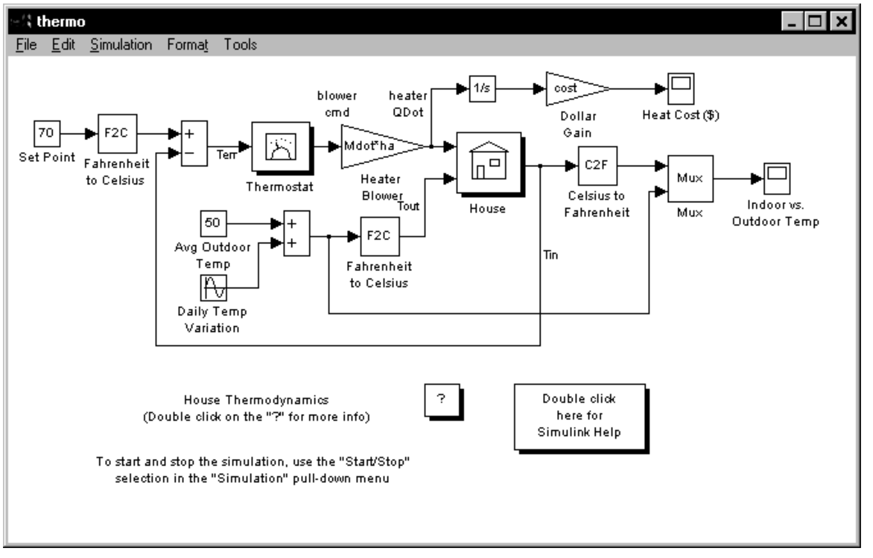
Quick Start

Running a Demo Model

An interesting demo program provided with Simulink models the thermodynamics of a house. To run this demo, follow these steps.

1 Start MATLAB. See your MATLAB documentation if you're not sure how to do this.

2 Run the demo model by typing `thermo` in the MATLAB command window. This command starts up Simulink and creates a model window that contains this model:



When you open the model, Simulink opens two Scope blocks, labeled Indoor vs. Outdoor Temp and Heat Cost ($).

3 To start the simulation, pull down the **Simulation** menu and choose the **Start** command. As the simulation runs, the indoor and outdoor temperatures appears in the Indoor vs. Outdoor Temp Scope block and the cumulative heating cost appears in the Heat Cost ($) Scope block.

4 To stop the simulation, choose the **Stop** command from the **Simulation** menu. If you want to explore other parts of the model, look over the suggestions in “Some Things to Try” on page 2–4.

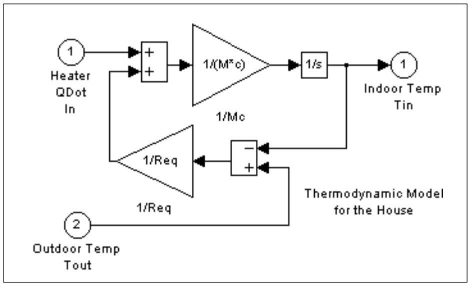
5 When you’re finished running the simulation, close the model by choosing **Close** from the **File** menu.

Description of the Demo

The demo models the thermodynamics of a house using a simple model. The thermostat is set to 70 degrees Fahrenheit and is affected by the outside temperature, which varies by applying a sine wave with amplitude of 15 degrees to a base temperature of 50 degrees. This simulates daily temperature fluctuations.

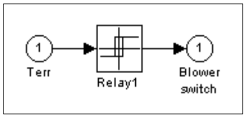
The model uses subsystems to simplify the model diagram and create reusable systems. A subsystem is a group of blocks that is represented by a Subsystem block. This model contains five subsystems: one named Thermostat, one named House, and three Temp Convert subsystems (two convert Fahrenheit to Celsius, one converts Celsius to Fahrenheit).

The internal and external temperatures are fed into the House subsystem, which updates the internal temperature. Double-click on the House block to see the underlying blocks in that subsystem.



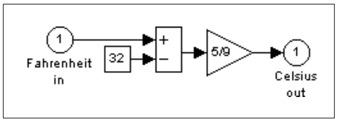
the House subsystem

The Thermostat subsystem models the operation of a thermostat, determining when the heating system is turned on and off. Double-click on the block to see the underlying blocks in that subsystem.



the Thermostat subsystem

Both the outside and inside temperatures are converted from Fahrenheit to Celsius by identical subsystems:



Fahrenheit to Celsius conversion (F2C)

When the heat is on, the heating costs are computed and displayed on the Heat Cost ($) Scope block. The internal temperature is displayed on the Indoor Temp Scope block.

Some Things to Try

Here are several things to try to see how the model responds to different parameters.

- Each Scope block contains a signal display area and controls that enable you to select the range of the signal displayed, zoom in on a portion of the signal, and perform other useful tasks. The horizontal axis represents time and the vertical axis represents the signal value. For more information about the Scope block, see Chapter 9.
- The Constant block labeled Set Point (at the top left of the model) sets the desired internal temperature. Open this block and reset the value to 80 degrees while the simulation is running. See how the indoor temperature and heating costs change. Also, adjust the outside temperature (the Avg Outdoor Temp block) and see how it affects the simulation.
- Adjust the daily temperature variation by opening the Sine Wave block labeled Daily Temp Variation and changing the **Amplitude** parameter.

What This Demo Illustrates

This demo illustrates several tasks commonly used when building models:

- Running the simulation involves specifying parameters and starting the simulation with the **Start** command, described in detail in Chapter 4.
- You can encapsulate complex groups of related blocks in a single block, called a subsystem. Creating subsystems is described in detail in Chapter 3.
- You can create a customized icon and design a dialog box for a block by using the masking feature, described in detail in Chapter 6. In the `thermo` model,

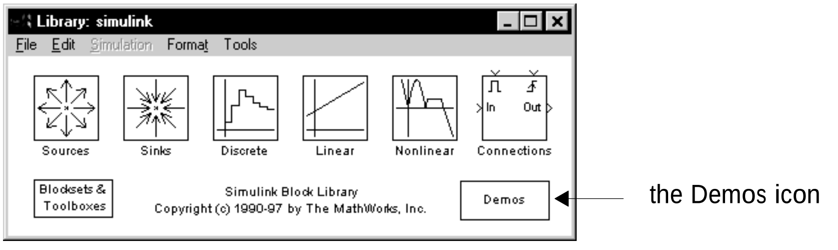
all Subsystem blocks have customized icons created using the masking feature.

- Scope blocks display graphic output much as an actual oscilloscope does. A Scope block displays its input signal. Scope blocks are described in detail in Chapter 9.

Other Useful Demos

Other demos illustrate useful modeling concepts. You can access these demos from the Simulink block library window:

1 Type `simulink` in the MATLAB command window. The Simulink block library window appears:

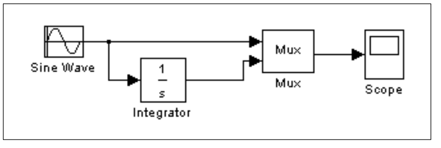


2 Double-click on the Demos icon. The MATLAB Demos window appears. This window contains several interesting sample models that illustrate useful Simulink features.

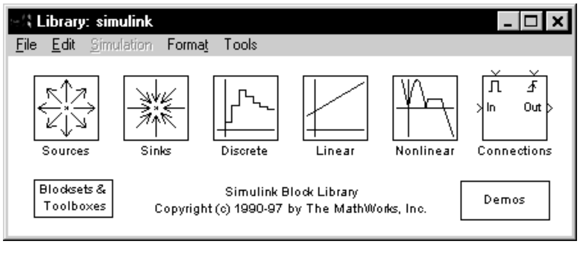
Building a Simple Model

This example shows you how to build a model using many of the model building commands and actions you will use to build your own models. The instructions for building this model in this section are brief. All of the tasks are described in more detail in the next chapter.

The model integrates a sine wave and displays the result, along with the sine wave. The block diagram of the model looks like this:



Type `simulink` in the MATLAB command window to display the Simulink block library and, if no other model window is open, a new untitled model window. The Simulink block library window looks like this:



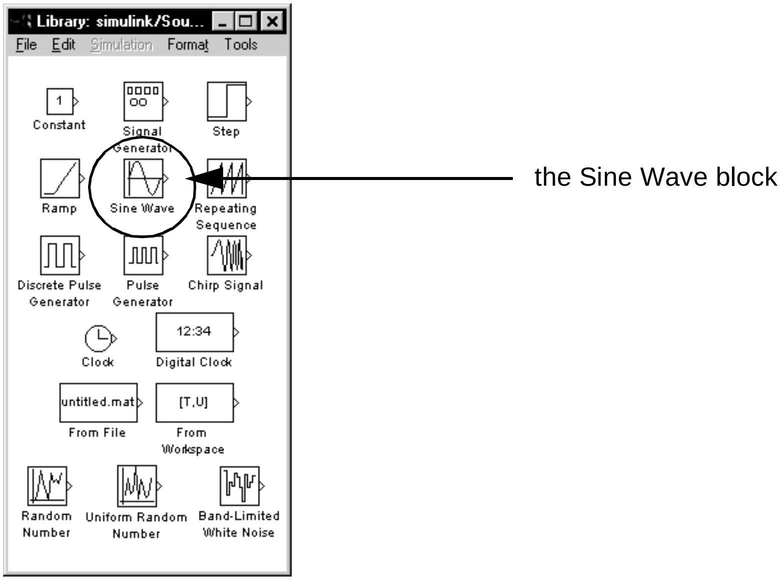
You might want to move the new model window to the right side of your screen so you can see its contents and the contents of block libraries at the same time.

In this model, you get blocks from these libraries:

- Sources library (the Sine Wave block)
- Sinks library (the Scope block)
- Linear library (the Integrator block)
- Connections library (the Mux block)

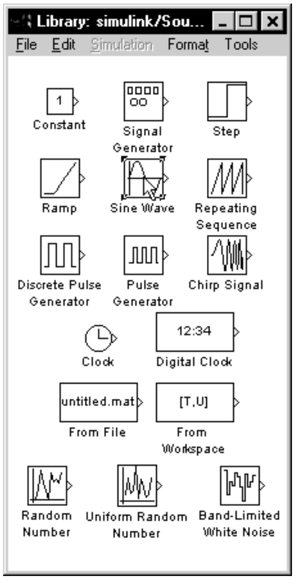
Open the Sources library to access the Sine Wave block. To open a block library, double-click on the library's icon. Simulink displays a window that contains all

the blocks in the library. In the Sources library, all blocks are signal sources. The Sources library window looks like this:

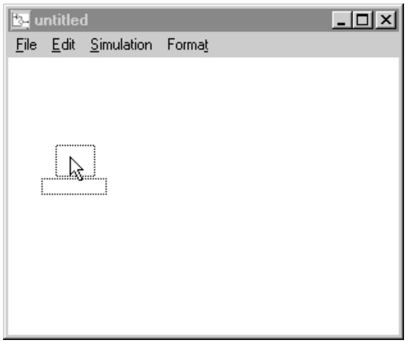


You add blocks to your model by copying them from a block library or from another model. For this exercise, you need to copy the Sine Wave block. To do this, position the cursor over the Sine Wave block, then press and hold down

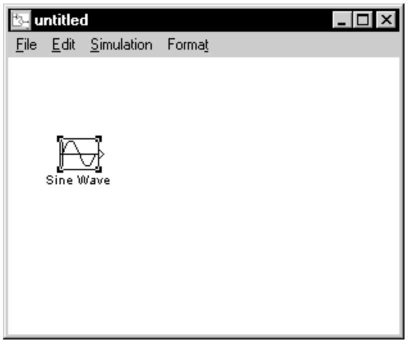
the mouse button. Notice how Simulink draws an outline around the block and its name.



Now, drag the block into the model window. As you move the block, you see the outline of the block and its name move with the pointer.

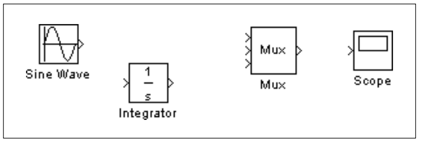


When the pointer is where you want the block to be in the model window, release the mouse button. A copy of the Sine Wave block is now in your model window.

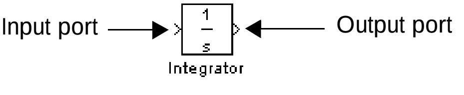


In the same way, copy the rest of the blocks into the model window. You can move a block from one place in the model window to another using the same dragging technique you used to copy the block. You can move a block a small amount by selecting the block, then pressing the arrow keys.

With all the blocks copied into the model window, the model should look something like this:

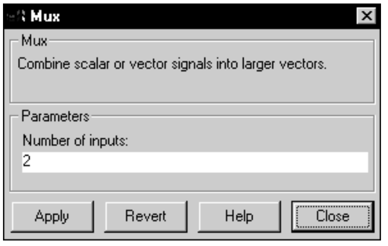


If you examine the block icons, you see an angle bracket on the right of the Sine Wave block and three on the left of the Mux block. The > symbol pointing out of a block is an *output port*; if the symbol points to a block, it is an *input port*. A signal travels out of an output port and into an input port of another block through a connecting line. When the blocks are connected, the port symbols disappear.

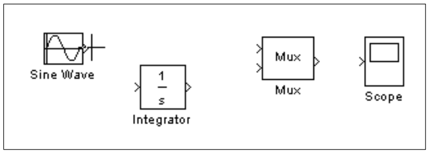


You might have noticed that the Mux block has three input ports but only two input signals. To change the number of input ports, open the Mux block's dialog box by double-clicking on the block. Change the **Number of inputs** parameter

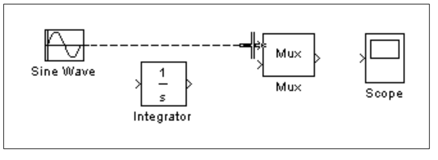
value to 2, then click on the **Close** button. Simulink adjusts the number of input ports.



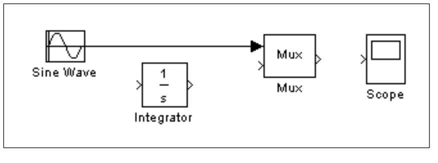
Now it's time to connect the blocks. Connect the Sine Wave block to the top input port of the Mux block: position the pointer over the output port on the right side of the Sine Wave block. Notice that the cursor shape changes to cross hairs.



Hold down the mouse button and move the cursor to the top input port of the Mux block. Notice that the line is dashed while the mouse button is down and that the cursor shape changes to double-lined cross hairs as it approaches the Mux block.



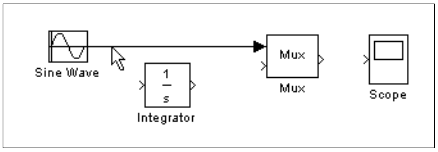
Now release the mouse button. The blocks are connected. You can also connect the line to the block by releasing the mouse button while the pointer is inside the icon. If you do, the line is connected to the input port closest to the cursor's position.



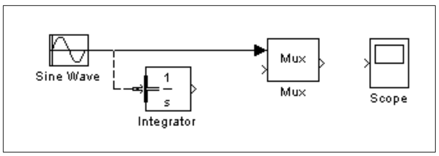
If you look again at the model on page 2–6, you'll notice that most of the lines connect output ports of blocks to input ports of other blocks. However, one line connects a *line* to the input port of another block. This line, called a *branch line*, connects the Sine Wave output to the Integrator block, and carries the same signal that passes from the Sine Wave block to the Mux block.

Drawing a branch line is slightly different from drawing the line you just drew. To weld a connection to an existing line, follow these steps:

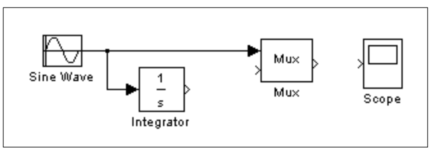
1 First, position the pointer *on the line* between the Sine Wave and the Mux block.



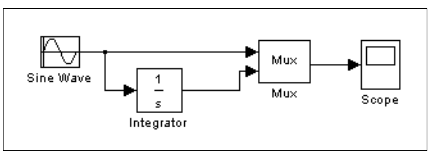
2 Press and hold down the **Ctrl** key on a Windows or X Windows system, or the **Option** key on a Macintosh. Press the mouse button, then drag the pointer to the Integrator block's input port or over the Integrator block itself.



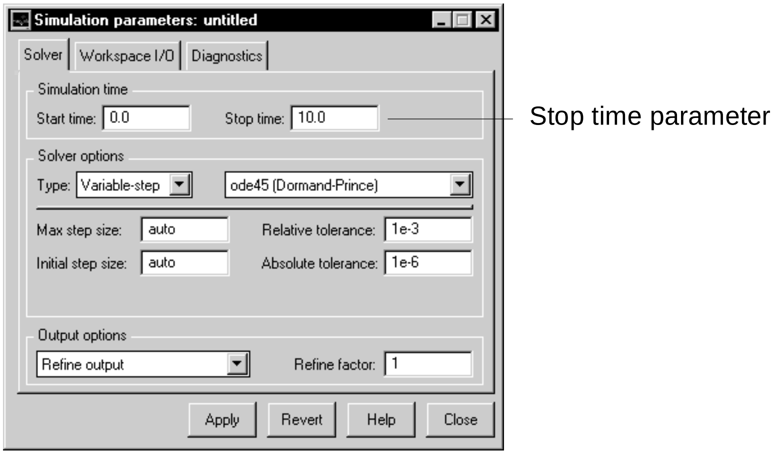
3 Release the mouse button. Simulink draws a line between the starting point and the Integrator block's input port.



Finish making block connections. When you're done, your model should look something like this:

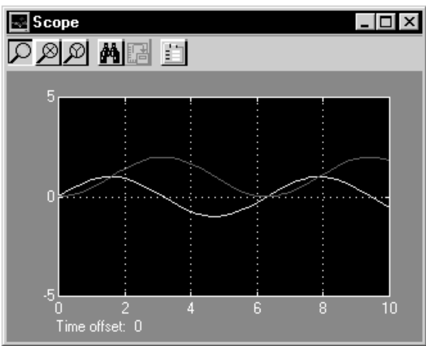


Now, open the Scope block to view the simulation output. Keeping the Scope window open, run the simulation for 10 seconds. First, set the simulation parameters by choosing **Parameters** from the **Simulation** menu. On the dialog box that appears, notice that the **Stop time** is set to 10.0 (its default value).



Close the **Simulation Parameters** dialog box by clicking on the **Close** button. Simulink applies the parameters and closes the dialog box.

Choose **Start** from the **Simulation** menu and watch the traces of the Scope block's input.



The simulation stops when it reaches the stop time specified in the **Simulation Parameters** dialog box or when you choose **Stop** from the **Simulation** menu.

To save this model, choose **Save** from the **File** menu and enter a filename and location. That file contains the description of the model.

To terminate Simulink and MATLAB, choose **Exit MATLAB** (on a Microsoft Windows system), **Quit MATLAB** (on an X Windows system), or **Quit** (on a Macintosh) from the **File** menu. You can also type `quit` in the MATLAB command window. If you want to leave Simulink but not terminate MATLAB, just close all Simulink windows.

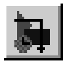
This exercise showed you how to perform some commonly used model-building tasks. These and other tasks are described in more detail in Chapter 3.

3

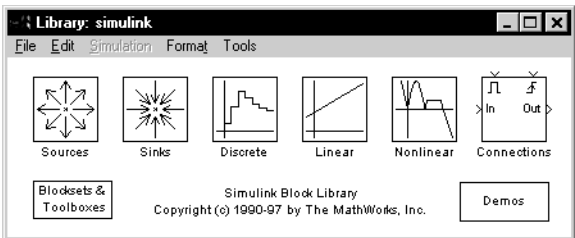
Creating a Model

Starting Simulink

To start Simulink, you must first start MATLAB. Consult your MATLAB documentation if you need more information. You can then start Simulink in two ways:

- Click on the Simulink icon  on the MATLAB toolbar.
- Enter the `simulink` command at the MATLAB prompt.

After MATLAB finishes processing the command, your desktop includes the MATLAB command window, a new empty model window (named `untitled`), and the Simulink block library window, shown below.



The Simulink block library window displays icons that represent the block libraries. You build models by copying blocks from block libraries into a model window (this procedure is described later in this chapter).

To run Simulink and work with your model, select commands from the Simulink menus or enter them in the MATLAB command window.

- Microsoft Windows and X Windows: a Simulink menu bar appears near the top of each model window. The menu commands apply to the contents of that window.
- Macintosh: the Simulink menu bar appears at the top of the desktop. The menu commands apply to the contents of the active window.

When you run a simulation and analyze its results, you can enter MATLAB commands in the MATLAB command window. Running a simulation is discussed in Chapter 4 and analyzing simulation results is discussed in Chapter 5.

Simulink Windows

Simulink uses separate windows to display a block library, a model, and graphical (scope) simulation output. These windows are not MATLAB figure windows and cannot be manipulated using Handle Graphics commands.

Simulink windows are sized to accommodate the most common screen resolutions available. If you have a monitor with exceptionally high or low resolution, you may find the window sizes too small or too large. If this is the case, resize the window and save the model to preserve the new window dimensions.

Creating a New Model

To create a new model, choose **New** from the **File** menu and select **Model**, or click on the Simulink icon on the MATLAB toolbar. You can move the window as you do other windows. Chapter 2 describes how to build a simple model. "Modeling Equations" on page 3–34 describes how to build systems that model equations.

Editing an Existing Model

To edit an existing model diagram, either

- Choose the **Open** command from the **File** menu, then choose or enter the model file name for the model you want to edit, or
- Enter the name of the model (without the `.mdl` extension) in the MATLAB command window. The model must be in the current directory or on the path.

Undoing a Command

You can cancel the effects of up to 101 consecutive operations by choosing **Undo** from the **Edit** menu. You can undo these operations:

- Adding or deleting a block.
- Adding or deleting a line.
- Adding or deleting a model annotation.
- Editing a block name.

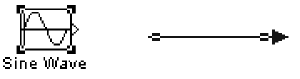
You can reverse the effects of an **Undo** command by choosing **Redo** from the **Edit** menu.

Selecting Objects

Many model building actions, such as copying a block or deleting a line, require that you first select one or more blocks and lines (objects).

Selecting One Object

To select an object, click on it. Small black square “handles” appear at the corners of a selected block and near the end points of a selected line. For example, the figure below shows a selected Sine Wave block and a selected line:



When you select an object by clicking on it, any other selected objects become deselected.

Selecting More than One Object

You can select more than one object either by selecting objects one at a time, by selecting objects located near each other using a bounding box, or by selecting the entire model.

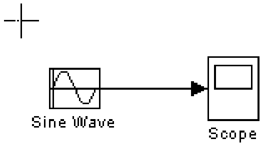
Selecting Multiple Objects One at a Time

To select more than one object by selecting each object individually, hold down the **Shift** key and click on each object to be selected. To deselect a selected object, click on the object again while holding down the **Shift** key.

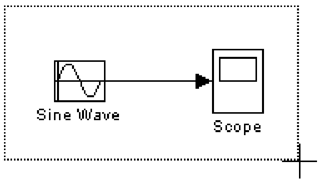
Selecting Multiple Objects Using a Bounding Box

An easy way to select more than one object in the same area of the window is to draw a bounding box around the objects.

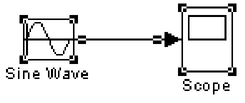
1 Define the starting corner of a bounding box by positioning the pointer at one corner of the box, then pressing and holding down the mouse button. Notice the shape of the cursor.



2 Drag the pointer to the opposite corner of the box. A dotted rectangle encloses the selected blocks and lines.



3 Release the mouse button. All blocks and lines at least partially enclosed by the bounding box are selected.



Selecting the Entire Model

To select all objects in the active window, choose **Select All** from the **Edit** menu. You cannot create a subsystem by selecting blocks and lines in this way; for more information, see “Creating Subsystems” on page 3–27.

Blocks

This section discusses how to perform useful model building actions involving blocks.

Copying and Moving Blocks from One Window to Another

As you build your model, you often copy blocks from Simulink block libraries or other libraries or models into your model window. To do this, follow these steps:

1. Open the appropriate block library or model window.
2. Drag the block you want to copy into the target model window. To drag a block, position the cursor over the block icon, then press and hold down the mouse button. Move the cursor into the target window, then release the mouse button.

You can also copy blocks by using the **Copy** and **Paste** commands from the **Edit** menu:

1. Select the block you want to copy.
2. Choose **Copy** from the **Edit** menu.
3. Make the target model window the active window.
4. Choose **Paste** from the **Edit** menu.

Simulink assigns a name to each copied block. If it is the first block of its type in the model, its name is the same as its name in the source window. For example, if you copy the Gain block from the Linear library into your model window, the name of the new block is Gain. If your model already contains a block named Gain, Simulink adds a sequence number to the block name (for example, Gain1, Gain2). You can rename blocks; see “Manipulating Block Names” on page 3–9.

When you copy a block, the new block inherits all the original block’s parameter values.

Simulink uses an invisible five-pixel grid to simplify the alignment of blocks. All blocks within a model snap to a line on the grid. You can move a block slightly up, down, left, or right by selecting the block and pressing the arrow keys.

You can copy or move blocks to compatible applications (such as word processing programs) using the **Copy**, **Cut**, and **Paste** commands. These commands copy only the graphic representation of the blocks, not their parameters.

Moving blocks from one window to another is similar to copying blocks, except that you hold down the **Shift** key while you select the blocks.

You can use the **Undo** command from the **Edit** menu to remove an added block.

Moving Blocks in a Model

To move a single block from one place to another in a model window, drag the block to a new location. Simulink automatically repositions lines connected to the moved block.

To move more than one block, including connecting lines:

1 Select the blocks and lines. If you need information about how to select more than one block, see "Selecting More than One Object" on page 3–4.
2 Drag the objects to their new location and release the mouse button.

Duplicating Blocks in a Model

You can duplicate blocks in a model by following this step:

- Microsoft Windows and X Windows: While holding down the **Ctrl** key, select the block with the left mouse button, then drag it to a new location. You can also do this by dragging the block using the right mouse button.
- Macintosh: While holding down the **Option** key, select the block and drag it to a new location.

Duplicated blocks have the same parameter values as the original blocks. Sequence numbers are added to the new block names.

Specifying Block Parameters

Certain aspects of a block's function are defined by the block's parameters. You can assign values to block parameters on the block's dialog box. Double-click on the block to open its dialog box. You can accept the displayed values or change them. You can also use the `set_param` command to change block parameters. See Chapter 11 for details.

The reference material that describes each block (in Chapter 9) shows the dialog box and describes the block parameters.

Deleting Blocks

To delete one or more blocks, select the blocks to be deleted and press the **Delete** or **Backspace** key. You can also choose **Clear** or **Cut** from the **Edit** menu. The **Cut** command writes the blocks into the clipboard, which enables you to paste them into a model. Using the **Delete** or **Backspace** key or the **Clear** command does not enable you to paste the block later.

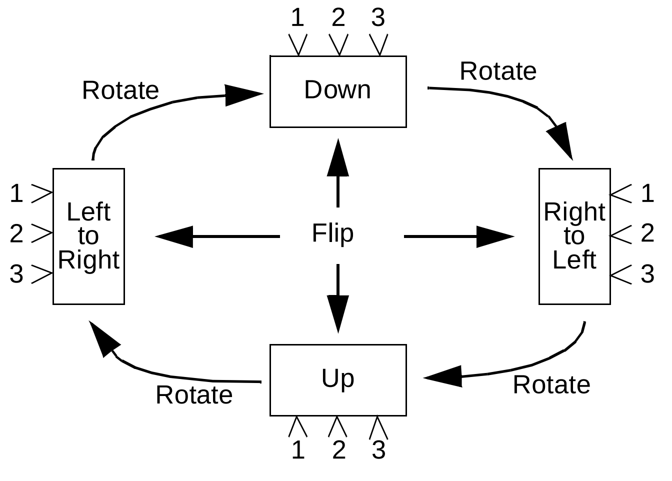
You can use the **Undo** command from the **Edit** menu to replace a deleted block.

Changing the Orientation of Blocks

By default, signals flow through a block from left to right. Input ports are on the left, and output ports are on the right. You can change the orientation of a block by choosing one of these commands from the **Format** menu:

- The **Flip Block** command rotates the block 180 degrees.
- The **Rotate Block** command rotates a block clockwise 90 degrees.

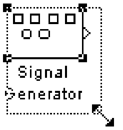
The figure below shows how Simulink orders ports after changing the orientation of a block using the **Rotate Block** and **Flip Block** menu items. The text in the blocks show their orientation.



Resizing Blocks

To change the size of a block, select it, then drag any of its selection handles. While you hold down the mouse button, a dotted rectangle shows the new block size. When you release the mouse button, the block is resized.

For example, the figure below shows a Signal Generator block being resized. The lower-right handle was selected and dragged to the cursor position. When the mouse button is released, the block takes its new size. This figure shows a block being resized:



Manipulating Block Names

All block names in a model must be unique and must contain at least one character. By default, block names appear below blocks whose ports are on the sides, and to the left of blocks whose ports are on the top and bottom, as this figure shows:

Changing Block Names

You can edit a block name in one of these ways:

- To replace the block name on a Microsoft Windows or X Windows system, click on the block name, then double-click or drag the cursor to select the entire name. Then, enter the new name. On a Macintosh, select the block name and replace the text.
- To insert characters, click between two characters to position the insertion point, then insert text.
- To replace characters, drag the mouse to select a range of text to replace, then enter the new text.

When you click the pointer someplace else in the model or take any other action, the name is accepted or rejected. If you try to change the name of a block

to a name that already exists or to a name with no characters, Simulink displays an error message.

You can modify the font used in a block name by selecting the block, then choosing the **Font** menu item from the **Format** menu. Select a font from the **Set Font** dialog box. This procedure also changes the font of text on the block icon.

You can cancel edits to a block name by choosing **Undo** from the **Edit** menu.

NOTE If you change the name of a library block, all links to that block will become unresolved.

Changing the Location of a Block Name

You can change the location of the name of a selected block in two ways:

- By dragging the block name to the opposite side of the block, or
- By choosing the **Flip Name** command from the **Format** menu. This command changes the location of the block name to the opposite side of the block.

For more information about block orientation, see "Changing the Orientation of Blocks" on page 3–8.

Changing Whether a Block Name Appears

To change whether the name of a selected block is displayed, choose a menu item from the **Format** menu:

- The **Hide Name** menu item hides a visible block name. When you select **Hide Name**, it changes to **Show Name** when that block is selected.
- The **Show Name** menu item shows a hidden block name.

Disconnecting Blocks

To disconnect a block from its connecting lines, hold down the **Shift** key, then drag the block to a new location.

Vector Input and Output

Almost all Simulink blocks accept scalar or vector inputs, generate scalar or vector outputs, and allow you to provide scalar or vector parameters. These blocks are referred to in this manual as being *vectorized*.

You can determine which lines in a model carry vector signals by choosing **Wide Vector Lines** from the **Format** menu. When this option is selected, lines that carry vectors are drawn thicker than lines that carry scalars. The figures in the next section show scalar and vector lines.

If you change your model after choosing **Wide Vector Lines**, you must explicitly update the display by choosing **Update Diagram** from the **Edit** menu. Starting the simulation also updates the block diagram display.

Block descriptions in Chapter 9 discuss the characteristics of block inputs, outputs, and parameters.

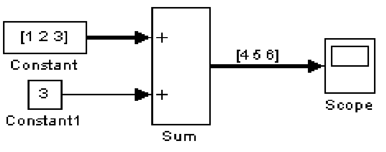
Scalar Expansion of Inputs and Parameters

Scalar expansion is the conversion of a scalar value into a vector of identical elements. Simulink applies scalar expansion to inputs and/or parameters for most blocks. Block descriptions in Chapter 9 indicate whether Simulink applies scalar expansion to a block's inputs and parameters.

Scalar Expansion of Inputs

When using blocks with more than one input port (such as the Sum or Relational Operator block), you can mix vector and scalar inputs. When you do this, the scalar inputs are expanded into vectors of identical elements whose widths are equal to the width of the vector inputs. (If more than one block input is a vector, they must have the same number of elements.)

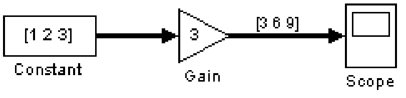
This model adds scalar and vector inputs. The input from block Constant1 is scalar expanded to match the size of the vector input from the Constant block. The input is expanded to the vector [3 3 3].



Scalar Expansion of Parameters

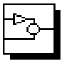
You can specify the parameters for vectorized blocks as either vectors or scalars. When you specify vector parameters, each parameter element is associated with the corresponding element in the input vector(s). When you specify scalar parameters, Simulink applies scalar expansion to convert them automatically into appropriately sized vectors.

This example shows that a scalar parameter (the Gain) is expanded to a vector of identically-valued elements to match the size of the block input, a three-element vector.



Using Drop Shadows

You can add a drop shadow to a block by selecting the block, then choosing **Show Drop Shadow** from the **Format** menu. When you select a block with a drop shadow, the menu item changes to **Hide Drop Shadow**. The figure below shows a Subsystem block with a drop shadow:



Libraries

This feature enables users to copy blocks into their models from external libraries and automatically update the copied blocks when the source blocks change. Using libraries allows users who develop their own block libraries, or who use those provided by others (such as blocksets), to ensure that their models automatically include the most recent versions of these blocks.

Terminology

It is important to understand the terminology used with this feature.

Library – A collection of library blocks. A library must be explicitly created using **New Library** from the **File** menu.

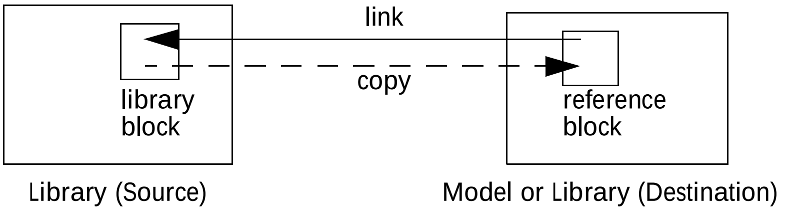
Library block – A block in a library.

Reference block – A copy of a library block.

Link – The connection between the reference block and its library block that allows Simulink to update the reference block when the library block changes.

Copy – The operation that creates a reference block from either a library block or another reference block.

This figure illustrates this terminology:



Creating a Library

To create a library, select **Library** from the **New** submenu of the **File** menu. Simulink displays a new window, labeled **Library: untitled**. If an untitled window already appears, a sequence number is appended.

You can create a library from the command line using this command:

```
new_system('newlib', 'Library')
```

This command creates a new library named `'newlib'`. To display the library, use the `open_system` command. These commands are described in Chapter 11.

The library must be named (saved) before you can copy blocks from it.

Modifying a Library

When you open a library, it is automatically locked and you cannot modify its contents. To unlock the library, select **Unlock Library** from the **Edit** menu. Closing the library window locks the library.

Copying a Library Block into a Model

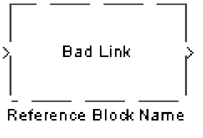
When you copy a library block into a model or another library, Simulink creates a link to the library block. The reference block is a copy of the library block. You can modify block parameters in the reference block but you cannot mask the block or, if it is masked, edit the mask. Also, you cannot set callback parameters for a reference block. If you look under the mask of a reference block, Simulink displays the underlying system for the library block.

The library and reference blocks are linked *by name*; that is, the reference block is linked to the specific block and library whose names are in effect at the time the copy is made.

If Simulink is unable to find either the library block or the source library on your MATLAB path when it attempts to update the reference block, the link becomes *unresolved*. Simulink issues an error message and displays these blocks using red dashed lines. The error message is:

```
Failed to find block "source-block-name"
in library "source-library-name"
referenced by block
"reference-block-path".
```

The unresolved reference block is displayed like this (colored red):



To fix a bad link, you must either:

- Delete the unlinked reference block and copy the library block back into your model, or
- Add the directory that contains the required library to the MATLAB path and select **Update Diagram** from the **Edit** menu, or
- Double-click on the reference block. On the dialog box that appears, correct the pathname and click on **Apply** or **Close**.

All blocks have a `LinkStatus` parameter that indicates whether the block is a reference block. The parameter can have these values:

- `'none'` indicates that the block is not a reference block.
- `'resolved'` indicates that the block is a reference block and that the link is resolved.
- `'unresolved'` indicates that the block is a reference block but that the link is unresolved.

Updating a Linked Block

Simulink updates reference blocks in a model or library at these times:

- When the model or library is loaded.
- When you select **Update Diagram** from the **Edit** menu or run the simulation in these circumstances:
  - When the library containing the library blocks is unlocked
  - When the model contains a reference block with an unresolved link
- When you query the `LinkStatus` parameter of a block using the `set_param` command (see the next section)
- When you use the `find_system` command

Breaking a Link to a Library Block

You can break the link between a reference block and its library block to cause the reference block to become a simple copy of the library block, unlinked to the library block. Changes to the library block no longer affect the block. Breaking links to library blocks enables you to transport a model as a stand-alone model, without the libraries.

To break the link between a reference block and its library block, select the block, then choose **Break Library Link** from the **Edit** menu. You can also break the link between a reference block and its library block from the command line by changing the value of the LinkStatus parameter to 'none' using this command:

```
set_param('refblock', 'LinkStatus', 'none')
```

You can save a system and break all links between reference blocks and library blocks using this command:

```
save_system('sys', 'newname', 'BreakLinks')
```

Finding the Library Block for a Reference Block

To find the source library and block linked to a reference block, select the reference block, then choose **Go To Library Link** from the **Edit** menu. If the library is open, Simulink selects the library block (displaying selection handles on the block) and makes the source library the active window. If the library is not open, Simulink opens it and selects the library block.

Getting Information About Library Blocks

Use the libinfo command to get information about reference blocks in a system. The format for the command is:

```
libdata = libinfo(sys)
```

where sys is the name of the system. The command returns a structure of size n-by-1, where n is the number of library blocks in sys. Each element of the structure has four fields:

- Block, the block path
- Library, the library name
- ReferenceBlock, the reference block path
- LinkStatus, the link status, either 'resolved' or 'unresolved'

Lines

Signals are carried on lines. Each line can carry a scalar or vector signal. A line can connect the output port of one block with the input port of another block. A line can also connect the output port of one block with input ports of many blocks by using branch lines.

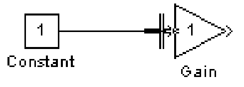
Drawing a Line Between Blocks

To connect the output port of one block to the input port of another block:

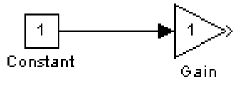
1 Position the cursor over the first block's output port. It is not necessary to position the cursor precisely on the port. The cursor shape changes to a cross hairs.

2 Press and hold down the mouse button.

3 Drag the pointer to the second block's input port. You can position the cursor on or near the port, or in the block. If you position the cursor in the block, the line is connected to the closest input port. The cursor shape changes to a double cross hairs.



4 Release the mouse button. Simulink replaces the port symbols by a connecting line with an arrow showing the direction of the signal flow. You can create lines either from output to input, or from input to output. The arrow is drawn at the appropriate input port, and the signal is the same.

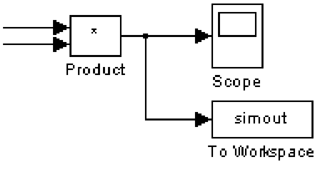


Simulink draws connecting lines using horizontal and vertical line segments. To draw a diagonal line, hold down the **Shift** key while drawing the line.

Drawing a Branch Line

A *branch line* is a line that starts from an existing line and carries its signal to the input port of a block. Both the existing line and the branch line carry the same signal. Using branch lines enables you to cause one signal to be carried to more than one block.

In this example, the output of the Product block goes to both the Scope block and the To Workspace block.



To add a branch line, follow these steps:

1 Position the pointer on the line where you want the branch line to start.
2 While holding down the **Ctrl** key (on a Microsoft Windows or X Windows system) or the **Option** key (on a Macintosh), press and hold down the mouse button.
3 Drag the pointer to the input port of the target block, then release the mouse button and the **Ctrl** or **Option** key.

On a Microsoft Windows or X Windows system, you can also use the right mouse button instead of holding down the **Ctrl** key while using the left mouse button.

Drawing a Line Segment

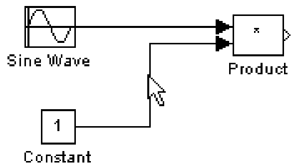
You may want to draw a line with segments exactly where you want them instead of where Simulink draws them. Or, you might want to draw a line before you copy the block to which the line is connected. You can do either by drawing line segments.

To draw a line segment, you draw a line that ends in an unoccupied area of the diagram. An arrow appears on the unconnected end of the line. To add another line segment, position the cursor over the end of the segment and draw another segment. Simulink draws the segments as horizontal and vertical lines. To draw diagonal line segments, hold down the **Shift** key while you draw the lines.

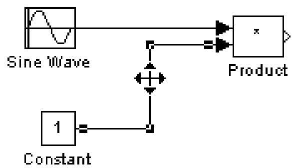
Moving a Line Segment

To move a line segment, follow these steps:

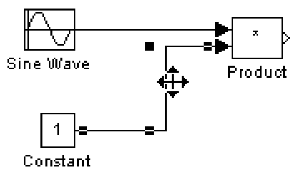
1 Position the pointer on the segment you want to move.



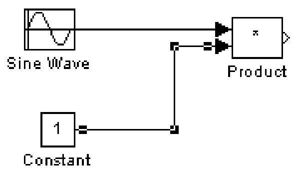
2 Press and hold down the mouse button.



3 Drag the pointer to the desired location.



4 Release the mouse button.

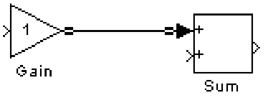


You cannot move the segments that are connected directly to block ports.

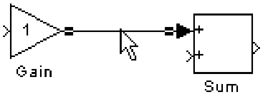
Dividing a Line into Segments

You can divide a line segment into two segments, leaving the ends of the line in their original locations. Simulink creates line segments and a vertex that joins them. To divide a line into segments, follow these steps:

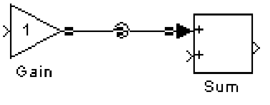
1 Select the line.



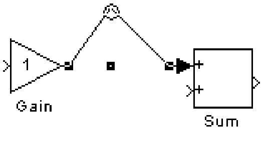
2 Position the pointer on the line where you want the vertex.



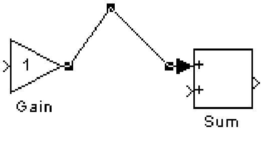
3 While holding down the **Shift** key, press and hold down the mouse button. The cursor shape changes to a circle that encloses the new vertex.



4 Drag the pointer to the desired location.



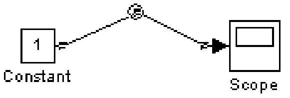
5 Release the mouse button and the **Shift** key.



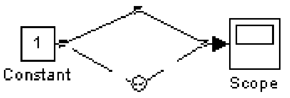
Moving a Line Vertex

To move a vertex of a line, follow these steps:

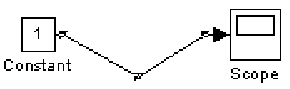
1. Position the pointer on the vertex, then press and hold down the mouse button. The cursor changes to a circle that encloses the vertex.



2. Drag the pointer to the desired location.



3. Release the mouse button.



Displaying Line Widths

You can display the widths of all lines in a model by turning on **Line Widths** from the **Format** menu. Simulink indicates the width of each signal at the block that originates the signal and the block that receives it.

When you start a simulation or update the diagram and Simulink detects a mismatch of input and output ports, it displays an error message and shows line widths in the model.

Signal Labels

You can label signals to annotate your model. Labels can appear above or below horizontal lines or line segments, and left or right of vertical lines or line segments. Labels can appear at either end, at the center, or in any combination of these locations.

Using Signal Labels

To create a signal label, double-click on the line segment and type the label at the insertion point. When you click on another part of the model, the label fixes its location.

NOTE When you create a signal label, take care to double-click *on* the line. If you click in an unoccupied area close to the line, you will create a model annotation instead.

To move a signal label, drag the label to a new location on the line. When you release the mouse button, the label fixes its position near the line.

To copy a signal label, hold down the **Ctrl** key (you can also use the **Option** key on a Macintosh) while dragging the label to another location on the line. When you release the mouse button, the label appears in the original and new location.

To edit a signal label, select it.

- To replace the label on a Microsoft Windows or X Windows system, click on the label, then double-click or drag the cursor to select the entire label. Then, enter the new label. On a Macintosh, select the label and enter the new text.
- To insert characters, click between two characters to position the insertion point, then insert text.
- To replace characters, drag the mouse to select a range of text to replace, then enter the new text.

To delete all occurrences of a signal label, delete all the characters in the label. When you click outside the label, the labels are deleted. To delete a single occurrence of the label, hold down the **Shift** key while you select the label, then press the **Delete** or **Backspace** key.

To change the font of a signal label, select the signal, choose **Font** from the **Format** menu, then select a font from the **Set Font** dialog box.

Signal Label Propagation

Signal label propagation is the automatic labeling of a line emitting from a connection block. The Connections library blocks that support signal label propagation are the Demux, Enable, From, Inport, Mux, Selector, and Subsystem blocks. The labeled signal must be on a line feeding a connecting block and the propagated signal must be on a line coming from the same connecting block or one associated with it.

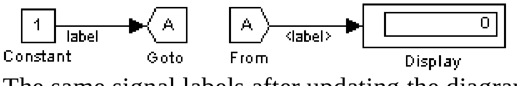
To propagate a signal label, create a signal label starting with the "<" character on the output of one of the listed connection blocks. When you run the simulation or update the diagram, the actual signal label appears, enclosed within angle brackets. The actual signal label is obtained by tracing back through the connection blocks until a signal label is encountered.

This example shows a model with a signal label and the propagated label both before and after updating the block diagram. In the first figure, the signal entering the Goto block is labeled `label` and the signal leaving the associated From block is labeled with a single <. The second figure shows the same model after choosing **Update Diagram** from the **Edit** menu:

1

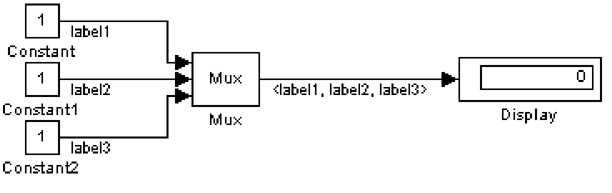
The signal label and propagated label before updating the diagram.

2



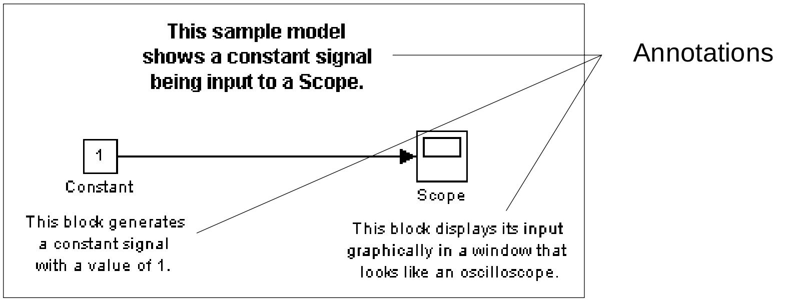
The same signal labels after updating the diagram.

In the next example, the propagated signal label shows the contents of a vector signal. This figure only shows the label *after* updating the diagram:



Annotations

Annotations provide textual information about a model. You can add an annotation to any unoccupied area of your block diagram. For example:



To create a model annotation, double-click on an unoccupied area of the block diagram. A small rectangle appears and the cursor changes to an insertion point. Start typing the annotation contents. Each line is centered within the rectangle that surrounds the annotation.

To move an annotation, drag it to a new location.

To edit an annotation, select it.

- To replace the annotation on a Microsoft Windows or X Windows system, click on the annotation, then double-click or drag the cursor to select it. Then, enter the new annotation. On a Macintosh, select the annotation, then enter new text.
- To insert characters, click between two characters to position the insertion point, then insert text.
- To replace characters, drag the mouse to select a range of text to replace, then enter the new text.

To delete an annotation, hold down the **Shift** key while you select the annotation, then press the **Delete** or **Backspace** key.

To change the font of all or part of an annotation, select the text in the annotation you want to change, then choose **Font** from the **Format** menu. Select a font and size from the dialog box.

Summary of Mouse and Keyboard Actions

These tables summarize the use of the mouse and keyboard to manipulate blocks, lines, and signal labels. LMB means press the left mouse button; CMB, the center mouse button; and RMB, the right mouse button.

The first table lists mouse and keyboard actions that apply to blocks.

Table 0-1: Mouse and Keyboard Actions that Manipulate Blocks

Task	Microsoft Windows	Macintosh	X Windows
Select one block	LMB	Mouse button	LMB
Select multiple blocks	**Shift** + LMB	**Shift** + mouse button	**Shift** + LMB; or CMB alone
Copy block from another window	Drag block	Drag block	Drag block
Move block	Drag block	Drag block	Drag block
Duplicate block	**Ctrl** + LMB and drag; or RMB and drag	**Option** + drag block	**Ctrl** + LMB and drag; or RMB and drag
Connect blocks	LMB	Mouse button	LMB
Disconnect block	**Shift** + drag block	**Shift** + drag block	**Shift** + drag block; or CMB and drag

The next table lists mouse and keyboard actions that apply to lines.

Table 0-2: Mouse and Keyboard Actions that Manipulate Lines

Task	Microsoft Windows	Macintosh	X Windows
Select one line	LMB	Mouse button	LMB
Select multiple lines	**Shift** + LMB	**Shift** + mouse button	**Shift** + LMB; or CMB alone
Draw branch line	**Ctrl** + drag line; or **RMB** and drag line	**Option** + drag line	**Ctrl** + drag line; or RMB + drag line

Table 0-2: Mouse and Keyboard Actions that Manipulate Lines (Continued)

Task	Microsoft Windows	Macintosh	X Windows
Route lines around blocks	**Shift** + draw line segments	**Shift** + draw line segments	**Shift** + draw line segments; or CMB and draw segments
Move line segment	Drag segment	Drag segment	Drag segment
Move vertex	Drag vertex	Drag vertex	Drag vertex
Create line segments	**Shift** + drag line	**Shift** + drag line	**Shift** + drag line; or CMB + drag line

The next table lists mouse and keyboard actions that apply to signal labels.

Table 0-3: Mouse and Keyboard Actions that Manipulate Signal Labels

Action	Microsoft Windows	Macintosh	X Windows
Create signal label	Double-click on line, then type label	Double-click on line, then type label	Double-click on line, then type label
Copy signal label	**Ctrl** + drag label	**Option** + drag label	**Ctrl** + drag label
Move signal label	Drag label	Drag label	Drag label
Edit signal label	Click in label, then edit	Click in label, then edit	Click in label, then edit
Delete signal label	**Shift** + click on label, then press **Delete**	**Shift** + click on label, then press **Delete**	**Shift** + click on label, then press **Delete**

The next table lists mouse and keyboard actions that apply to annotations.

Table 0-4: Mouse and Keyboard Actions that Manipulate Annotations

Action	Microsoft Windows	Macintosh	X Windows
Create annotation	Double-click in diagram, then type text	Double-click in diagram, then type text	Double-click in diagram, then type text
Copy annotation	**Ctrl** + drag label	**Option** + drag label	**Ctrl** + drag label
Move annotation	Drag label	Drag label	Drag label
Edit annotation	Click in text, then edit	Click in text, then edit	Click in text, then edit
Delete annotation	**Shift** + select annotation, then press **Delete**	**Shift** + select annotation, then press **Delete**	**Shift** + select annotation, then press **Delete**

Creating Subsystems

As your model increases in size and complexity, you can simplify it by grouping blocks into subsystems. Using subsystems has these advantages:

- It helps reduce the number of blocks displayed in your model window.
- It allows you to keep functionally related blocks together.
- It enables you to establish a hierarchical block diagram, where a Subsystem block is on one layer and the blocks that make up the subsystem are on another.

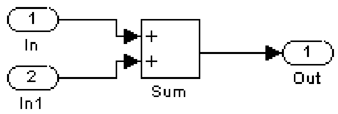
You can create a subsystem in two ways:

- Add a Subsystem block to your model, then open that block and add the blocks it contains to the subsystem window.
- Add the blocks that make up the subsystem, then group those blocks into a subsystem.

Creating a Subsystem by Adding the Subsystem Block

To create a subsystem before adding the blocks it contains, add a Subsystem block to the model, then add the blocks that make up the subsystem:

1 Copy the Subsystem block from the Connections library into your model.
2 Open the Subsystem block by double-clicking on it.
3 In the empty Subsystem window, create the subsystem. Use Inport blocks to represent input from outside the subsystem and Outport blocks to represent external output. For example, the subsystem below includes a Sum block and Inport and Outport blocks to represent input to and output from the subsystem:

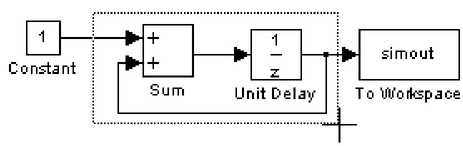


Creating a Subsystem by Grouping Existing Blocks

If your model already contains the blocks you want to convert to a subsystem, you can create the subsystem by grouping those blocks:

1 Enclose the blocks and connecting lines you want to include in the subsystem within a bounding box. You cannot specify the blocks to be grouped by selecting them individually or by using the **Select All** command. For more information, see "Selecting Multiple Objects Using a Bounding Box" on page 3–5.

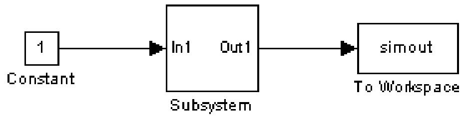
   For example, this figure shows a model that represents a counter. The Sum and Unit Delay blocks are selected within a bounding box:



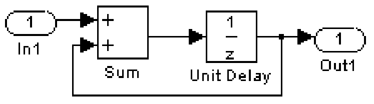
   When you release the mouse button, the two blocks and all the connecting lines are selected.

2 Choose **Create Subsystem** from the **Edit** menu. Simulink replaces the selected blocks with a Subsystem block. This figure shows the model after

choosing the **Create Subsystem** command (and resizing the Subsystem block so the port labels are readable):



If you open the Subsystem block, Simulink displays the underlying system, as shown below. Notice that Simulink adds Inport and Outport blocks to represent input from and output to blocks outside the subsystem:



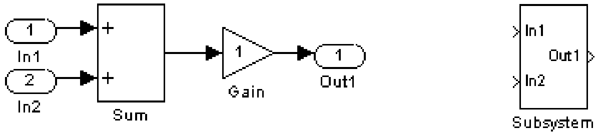
As with all blocks, you can change the name of the Subsystem block. Also, you can customize the icon and dialog box for the block using the masking feature, described in Chapter 6.

Labeling Subsystem Ports

Simulink labels ports on a Subsystem block. The labels are the names of Inport and Outport blocks that connect the subsystem to blocks outside the subsystem through these ports.

You can hide the port labels by selecting the Subsystem block, then choosing **Hide Port Labels** from the **Format** menu. You can also hide one or more port labels by selecting the appropriate Inport or Outport block in the subsystem and choosing **Hide Name** from the **Format** menu.

This figure shows two models. The subsystem on the left contains two Inport blocks and one Outport block. The Subsystem block on the right shows the labeled ports:



Subsystem with Inport and Outport blocks

Subsystem with labeled ports

Using Callback Routines

You can define MATLAB expressions that execute when the block diagram or a block is acted upon in a particular way. These expressions, called *callback routines*, are associated with block or model parameters. For example, the callback associated with a block's `OpenFcn` parameter is executed when the model user double-clicks on that block's name or path changes.

To define callback routines and associate them with parameters, use the `set_param` command, described on page 11–22.

For example, this command evaluates the variable `testvar` when the user double-clicks on the Test block in `mymodel`:

```
set_param('mymodel/Test', 'OpenFcn', testvar)
```

You can examine the `clutch` system (`clutch.mdl`) for routines associated with many model callbacks.

These tables list the parameters for which you can define callback routines, and indicate when those callback routines are executed. Routines that are executed before or after actions take place occur immediately before or after the action.

Table 3-1: Model Callback Parameters

Parameter	When Executed
`CloseFcn`	Before the block diagram is closed.
`PostLoadFcn`	After the model is loaded. Defining a callback routine for this parameter might be useful for generating an interface that requires that the model has already been loaded.
`PostSaveFcn`	After the model is saved.
`PreLoadFcn`	Before the model is loaded. Defining a callback routine for this parameter might be useful for loading variables used by the model.
`PreSaveFcn`	Before the model is saved.
`StartFcn`	Before the simulation starts.
`StopFcn`	After the simulation stops. Output is written to workspace variables and files before the `StopFcn` is executed.

Table 3-2: Block Callback Parameters

Parameter	When Executed
`CloseFcn`	When the block is closed using the `close_system` command.
`CopyFcn`	After a block is copied. The callback is recursive for Subsystem blocks (that is, if you copy a Subsystem block that contains a block for which the `CopyFcn` parameter is defined, the routine is also executed). The routine is also executed if an `add_block` command is used to copy the block.
`DeleteFcn`	Before a block is deleted. This callback is recursive for Subsystem blocks.
`InitFcn`	Before the block diagram is compiled and before block parameters are evaluated.
`LoadFcn`	After the block diagram is loaded. This callback is recursive for Subsystem blocks.
`ModelCloseFcn`	Before the block diagram is closed. This callback is recursive for Subsystem blocks.
`NameChangeFcn`	After a block's name and/or path changes. When a Subsystem block's path is changed, it recursively calls this function for all blocks it contains after calling its own `NameChangeFcn` routine.
`OpenFcn`	When the block is opened. This parameter is generally used with Subsystem blocks. The routine is executed when you double-click on the block or when an `open_system` command is called with the block as an argument. The `OpenFcn` parameter overrides the normal behavior associated with opening a block, which is to display the block's dialog box or to open the subsystem.
`ParentCloseFcn`	Before closing a subsystem containing the block or when the block is made part of a new subsystem using the **Create Subsystem** command.
`PreSaveFcn`	Before the block diagram is saved. This callback is recursive for Subsystem blocks.
`PostSaveFcn`	After the block diagram is saved. This callback is recursive for Subsystem blocks.

Table 3-2: Block Callback Parameters (Continued)

Parameter	When Executed
`StartFcn`	After the block diagram is compiled and before the simulation starts.
`StopFcn`	At any termination of the simulation.
`UndoDelete`	When a block delete is undone.

Tips for Building Models

Here are some model-building hints you might find useful.

- Memory issues

  In general, the more memory, the better Simulink performs.

- Using hierarchy

  More complex models often benefit from adding the hierarchy of subsystems to the model. Grouping blocks simplifies the top level of the model and can make it easier to read and understand the model. For more information, see “Creating Subsystems” on page 3–27. The Model Browser, described on page 3–42, provides useful information about complex models.

- Cleaning up models

  Well organized and documented models are easier to read and understand. Signal labels and model annotations can help describe what is happening in a model. For more information, see “Signal Labels” on page 3–22 and “Annotations” on page 3–24.

- Modeling strategies

  If several of your models tend to use the same blocks, you might find it easier to save these blocks in a model. Then, when you build new models, just open this model and copy the commonly used blocks from it. You can create a block library by placing a collection of blocks into a system and saving the system. You can then access the system by typing its name in the MATLAB command window.

  Generally, when building a model, design it first on paper, then build it using the computer. Then, when you start putting the blocks together into a model, add the blocks to the model window before adding the lines that connect them. This way, you can reduce how often you need to open block libraries.

Modeling Equations

One of the most confusing issues for new Simulink users is how to model equations. Here are some examples that may improve your understanding of how to model equations.

Converting Celsius to Fahrenheit

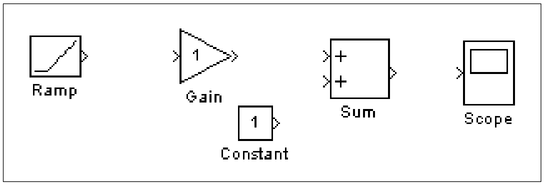
To model the equation that converts Celsius temperature to Fahrenheit:

$T_F = 9/5(T_C) + 32$

First, consider the blocks needed to build the model:

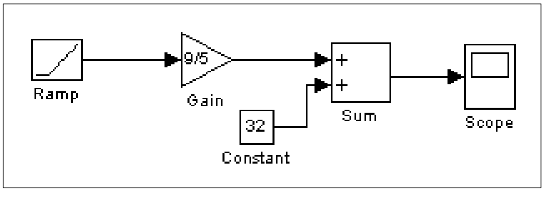
- A Ramp block to input the temperature signal, from the Sources library
- A Constant block, to define a constant of 32, also from the Sources library
- A Gain block, to multiply the input signal by 9/5, from the Linear library
- A Sum block, to add the two quantities, also from the Linear library
- A Scope block to display the output, from the Sinks library

Next, gather the blocks into your model window:



Assign parameter values to the Gain and Constant blocks by opening (double-clicking on) each block and entering the appropriate value. Then, click on the **Close** button to apply the value and close the dialog box.

Now, connect the blocks.



The Ramp block inputs Celsius temperature. Open that block and change the **Initial output** parameter to 0. The Gain block multiplies that temperature by the constant 9/5. The Sum block adds the value 32 to the result and outputs the Fahrenheit temperature.

Open the Scope block to view the output. Now, choose **Start** from the **Simulation** menu to run the simulation. The simulation will run for 10 seconds.

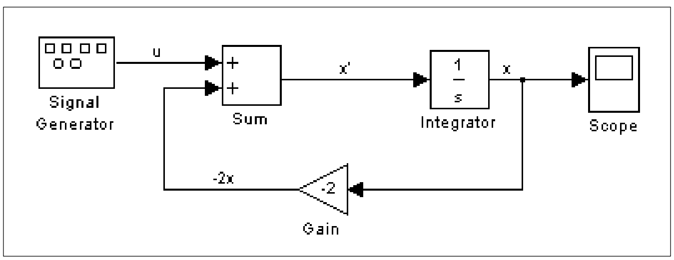
Modeling a Simple Continuous System

To model the differential equation

$$x'(t) = -2x(t) + u(t)$$

where *u(t)* is a square wave with an amplitude of 1 and a frequency of 1 rad/sec. The Integrator block integrates its input, x', to produce x. Other blocks needed in this model include a Gain block and a Sum block. To generate a square wave, use a Signal Generator block and select the Square Wave form but change the default units to radians/sec. Again, view the output using a Scope block. Gather the blocks and define the Gain.

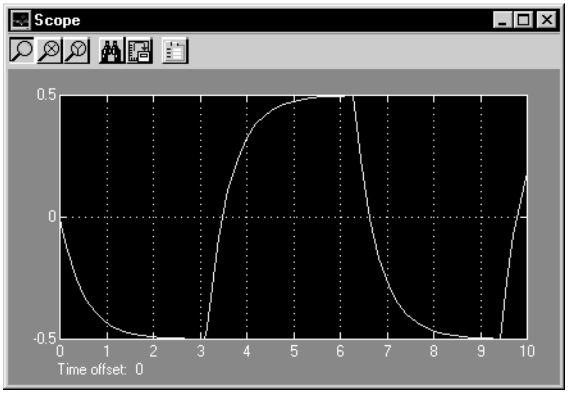
In this model, to reverse the direction of the Gain block, select the block, then use the **Flip Block** command from the **Format** menu. Also, to create the branch line from the output of the Integrator block to the Gain block, hold down the **Ctrl** key (on a Microsoft Windows or X Windows system) or the **Option** key (on a Macintosh) while drawing the line. For more information, see “Drawing a Branch Line” on page 3–18. Now you can connect all the blocks. These procedures are discussed earlier in this chapter.



An important concept in this model is the loop that includes the Sum block, the Integrator block, and the Gain block. In this equation, x is the output of the

Integrator block. It is also the input to the blocks that compute x', on which it is based. This relationship is implemented using a loop.

The Scope displays x at each time step. For a simulation lasting 10 seconds, the output looks like this:



The equation you modeled in this example can also be expressed as a transfer function. The model uses the Transfer Fcn block, which accepts u as input and outputs x. So, the block implements x/u. If you substitute sx for x' in the equation above:

$$sx = -2x + u$$

Solving for x gives

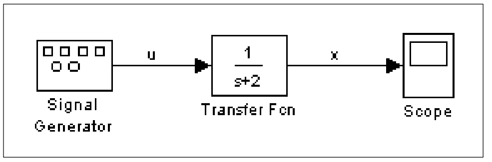
$$x = u/(s + 2)$$

Or,

$$x/u = 1/(s + 2)$$

The Transfer Fcn block uses parameters to specify the numerator and denominator coefficients. In this case, the numerator is `1` and the denominator is `s+2`. Specify both terms as vectors of coefficients of successively decreasing

powers of `s`. In this case the numerator is `[1]` (or just `1`) and the denominator is `[1 2]`. The model now becomes quite simple:



The results of this simulation are identical to those of the previous model.

Saving a Model

You can save a model by choosing either the **Save** or **Save As** command from the **File** menu. Simulink saves the model by generating a specially formatted file called the *model file* (with the `.mdl` extension) that contains the block diagram and block properties. The format of the model file is described in Appendix B.

If you are saving a model for the first time, use the **Save** command to provide a name and location to the model file. Model file names must start with a letter and can contain no more than 31 letters, numbers, and underscores.

If you are saving a model whose model file was previously saved, use the **Save** command to replace the file's contents or the **Save As** command to save the model with a new name or location.

Printing a Block Diagram

You can print a block diagram by selecting **Print** from the **File** menu (on a Microsoft Windows or Macintosh system) or by using the `print` command in the MATLAB command window (on all platforms).

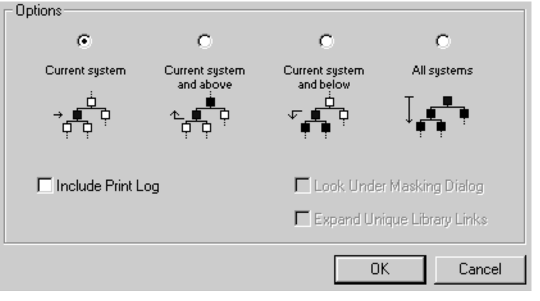
On a Microsoft Windows system, the **Print** menu item prints the block diagram in the current window. On a Macintosh system, the **Print** menu item prints the system in the active window.

Print Dialog Box

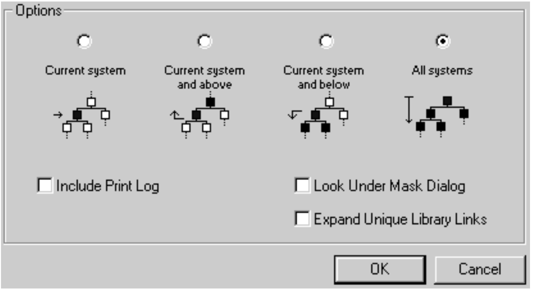
When you select the **Print** menu item, the **Print** dialog box appears. The **Print** dialog box enables you to selectively print systems within your model. Using the dialog box, you can:

- Print the current system only
- Print the current system and all systems above it in the model hierarchy
- Print the current system and all systems below it in the model hierarchy, with the option of looking into the contents of masked and library blocks
- Print all systems in the model, with the option of looking into the contents of masked and library blocks

The portion of the **Print** dialog box that supports selective printing is similar on supported platforms. This figure shows how it looks on a Microsoft Windows system. In this figure, only the current system is to be printed:



When you select either the **Current system and below** or **All systems** option, two check boxes become enabled. In this figure, **All systems** is selected.



Selecting the **Look Under Mask Dialog** check box prints the contents of masked subsystems when encountered at or below the level of the current block. When printing all systems, the top-level system is considered the current block so Simulink looks under any masked blocks encountered.

Selecting the **Expand Unique Library Links** check box prints the contents of library blocks when those blocks are systems. Only one copy is printed regardless of how many copies of the block are contained in the model. For more information about libraries, see “Libraries” on page 3-13.

The print log lists the blocks and systems printed. To print the print log, select the **Include Print Log** check box.

Print Command

The format of the `print` command is:

```
print –ssys –device filename
```

sys is the name of the system to be printed. The system name must be preceded by the `s` switch identifier and is the only required argument. *sys* must be open or must have been open during the current session. If the system name contains spaces or takes more than one line, you need to specify the name as a string. See the examples below.

device specifies a device type. For a list and description of device types, see *Using MATLAB Graphics*.

filename is the PostScript file to which the output is saved. If `filename` exists, it is replaced. If `filename` does not include an extension, an appropriate one is appended.

For example, this command prints a system named `untitled`:

```
print –suntitled
```

This command prints the contents of a subsystem named Sub1 in the current system:

```
print –sSub1
```

This command prints the contents of a subsystem named Requisite Friction:

```
print (['–sRequisite Friction'])
```

The next example prints a system named `Friction Model`, a subsystem whose name appears on two lines. The first command assigns the newline character to a variable; the second prints the system.

```
cr = sprintf('\n');
print (['–sFriction' cr 'Model'])
```

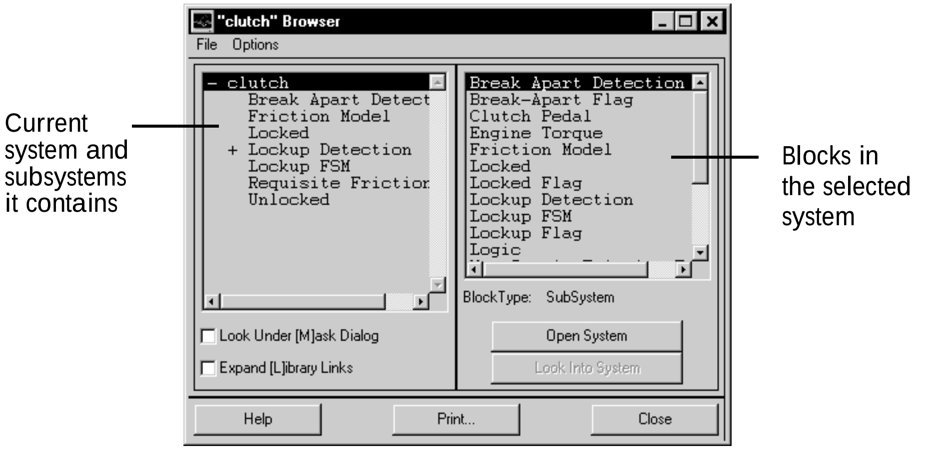
You cannot control the size of the system when print output is sent directly to the printer. If the diagram is larger than the page size, Simulink reduces it to fit the page. To control the size of the print output, direct output to an EPS file or to a bitmap, then manipulate its size using a word processing program.

The Model Browser

The Model Browser enables you to:

- Navigate a model hierarchically.
- Open systems in a model directly.
- Determine the blocks contained in a model.

To open the Browser, select **Show Browser** from the **File** menu. The Browser window appears, displaying information about the current model. This figure shows the Browser window displaying the contents of the `clutch` system:



Contents of the Browser Window

The Browser window consists of:

- The systems list. The list on the left contains the current system and the subsystems it contains, with the current system selected.
- The blocks list. The list on the right contains the names of blocks in the selected system. Initially, this window displays blocks in the top-level system.
- The **File** menu, which contains the **Print**, **Close Model**, and **Close Browser** menu items.

- The **Options** menu, which contains these menu items: **Open System**, **Look Into System**, **Display Alphabetical/Hierarchical List**, **Expand All**, **Look Under Mask Dialog**, and **Expand Library Links**.
- **Options** check boxes and buttons: **Look Under [M]ask Dialog** and **Expand [L]ibrary Links** check boxes, and **Open System** and **Look Into System** buttons. By default, Simulink does not display contents of masked blocks and blocks that are library links. These check boxes enable you to override the default.
- The block type of the selected block.
- Dialog box buttons: **Help**, **Print**, and **Close**.

Interpreting List Contents

Simulink identifies masked blocks, reference blocks, blocks with defined `OpenFcn` parameters, and systems that contain subsystems using these symbols before a block or system name:

- A plus sign (+) before a system name in the systems list indicates that the system is expandable, which means that it has systems beneath it. Double-click on the system name to expand the list and display its contents in the blocks list. When a system is expanded, a minus sign (–) appears before its name.
- [M] indicates that the block is masked, having either a mask dialog box or a mask workspace. For more information about masking, see Chapter 6.
- [L] indicates that the block is a reference block. For more information, see “Libraries” on page 3-13.
- [O] indicates that an open function (`OpenFcn`) callback is defined for the block. For more information about block callbacks, see “Using Callback Routines” on page 3-30.
- [S] indicates that the system is a Stateflow™ block.

Opening a System

You can open any block or system whose name appears in the blocks list. To open a system:

1 In the systems list, select by single-clicking on the name of the parent system that contains the system you want to open. The parent system's contents appear in the blocks list.

2 Depending on whether the system is masked, linked to a library block, or has an open function callback, you open it as follows:

   If the system has no symbol to its left, double-click on its name or select its name and click on the **Open System** button.

   If the system has an [M] or [O] before its name, select the system name and click on the **Look Into System** button.

Looking into a Masked System or a Linked Block

By default, the Browser considers masked systems (identified by [M]) and linked blocks (identified by [L]) as blocks and not subsystems. If you click on **Open System** while a masked system or linked block is selected, the Browser displays the system or block's dialog box (**Open System** works the same way as double-clicking on the block in a block diagram). Similarly, if the block's `OpenFcn` callback parameter is defined, clicking on **Open System** while that block is selected executes the callback function.

You can direct the Browser to look beyond the dialog box or callback function by selecting the block in the blocks list, then clicking on **Look Into System**. The Browser displays the underlying system or block.

Displaying List Contents Alphabetically

By default, the systems list indicates the hierarchy of the model. Systems that contain systems are preceded with a plus sign (+). When those systems are expanded, the Browser displays a minus sign (–) before their names. To display systems alphabetically, select the **Display Alphabetical List** menu item on the **Options** menu.

Ending a Simulink Session

Terminate a Simulink session by closing all Simulink windows.

Terminate a MATLAB session by choosing one of these commands from the **File** menu:

- On a Microsoft Windows system: **Exit MATLAB**
- On a Macintosh system: **Quit**
- On an X Windows system: **Quit MATLAB**

4

Running a Simulation

Introduction

You can run a simulation either by using Simulink menu commands or by entering commands in the MATLAB command window.

Many users use menu commands while they develop and refine their models, then enter commands in the MATLAB command window to run the simulation in “batch” mode.

Using Menu Commands

Running a simulation using menu commands is easy and interactive. These commands let you select an ODE solver and define simulation parameters without having to remember command syntax. An important advantage is that you can perform certain operations interactively while a simulation is running:

- You can modify many simulation parameters, including the stop time, the solver, and the maximum step size.
- You can change the solver.
- You can simulate another system at the same time.
- You can click on a line to see the signal carried on that line on a floating (unconnected) Scope or Display block.
- You can modify the parameters of a block, as long as you do not cause a change in:
  - The number of states, inputs, or outputs
  - The sample time
  - The number of zero crossings
  - The vector length of any block parameters
  - The length of the internal block work vectors

You cannot make changes to the structure of the model, such as adding or deleting lines or blocks, during a simulation. If you need to make these kinds of changes, you need to stop the simulation, make the change, then start the simulation again to see the results of the change.

Running a Simulation from the Command Line

Running a simulation from the command line has these advantages over running a simulation using menu commands:

- You can simulate M-file and MEX-file models, as well as Simulink block diagram models.
- You can run a simulation from an M-file, allowing simulation and block parameters to be changed iteratively.

For more information, see "Running a Simulation from the Command Line" on page 4–21.

Running a Simulation Using Menu Commands

This section discusses how to use Simulink menu commands and the **Simulation Parameters** dialog box to run a simulation.

Setting Simulation Parameters and Choosing the Solver

You set the simulation parameters and select the solver by choosing **Parameters** from the **Simulation** menu. Simulink displays the **Simulation Parameters** dialog box, which uses three “pages” to manage simulation parameters:

- The **Solver** page allows you to set the start and stop times, choose the solver and specify solver parameters, and choose some output options.
- The **Workspace I/O** page manages input from and output to the MATLAB workspace.
- The **Diagnostics** page allows you to select the level of warning messages displayed during a simulation.

Each page of the dialog box, including the parameters you set on the page, is discussed in detail in “The Simulation Parameters Dialog Box” on page 4–6.

You can specify parameters as valid MATLAB expressions, consisting of constants, workspace variable names, MATLAB functions, and mathematical operators.

Applying the Simulation Parameters

After you have set the simulation parameters and selected the solver, you are ready to apply them to your model. Press the **Apply** button on the bottom of the dialog box to apply the parameters to the model. To apply the parameters and close the dialog box, press the **Close** button.

Starting the Simulation

After you have applied the solver and simulation parameters to your model, you are ready to run the simulation. Select **Start** from the **Simulation** menu to run the simulation. You can also use the keyboard shortcut, **Ctrl-T** on a Microsoft Windows or X Windows system, or **-T** on a Macintosh. When you select **Start**, the menu item changes to **Stop**.

Your computer beeps to signal the completion of the simulation.

NOTE A common mistake that new Simulink users make is to start a simulation while the Simulink block library is the active window. Make sure your model window is the active window before starting a simulation.

To stop a simulation, choose **Stop** from the **Simulation** menu. The keyboard shortcut for stopping a simulation is **Ctrl-T** on a Microsoft Windows or X Windows system, or **-T** on a Macintosh, the same as for starting a simulation.

You can suspend a running simulation by choosing **Pause** from the **Simulation** menu. When you select **Pause**, the menu item changes to **Continue**. You proceed with a suspended simulation by choosing **Continue**.

If the model includes any blocks that write output to a file or to the workspace, or if you select output options on the **Simulation Parameters** dialog box, Simulink writes the data when the simulation is terminated or suspended.

The Simulation Parameters Dialog Box

This section discusses the simulation parameters, which you specify either on the **Simulation Parameters** dialog box or using the `sim` and `simset` commands (described later in this chapter). Parameters are described as they appear on the dialog box pages.

This table summarizes the actions performed by the dialog box buttons, which appear on the bottom of each dialog box page.

Table 4-1: Simulation Parameters Dialog Box Buttons

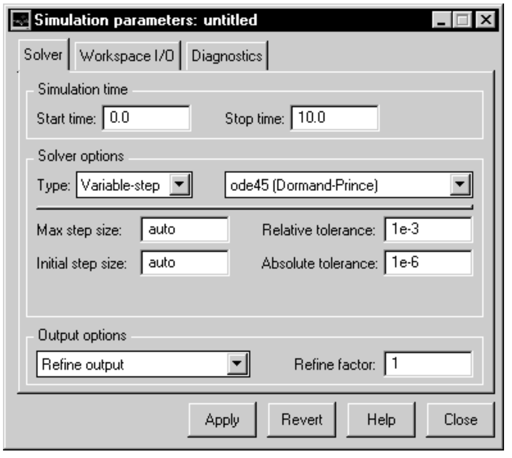
Button	Action
Apply	Applies the current parameter values and keeps the dialog box open. During a simulation, the parameter values are applied immediately.
Revert	Changes the parameter values back to the values they had when the dialog box was most recently opened and applies the parameters.
Help	Displays help text for the dialog box page.
Close	Applies the parameter values and closes the dialog box. During a simulation, the parameter values are applied immediately.

The Solver Page

The **Solver** page appears when you first choose **Parameters** from the **Simulation** menu or when you select the **Solver** tab.

The **Solver** page allows you to

- Set the simulation start and stop times.
- Choose the solver and specify its parameters.
- Select output options.



Simulation Time

You can change the start time and stop time for the simulation by entering new values in the **Start time** and **Stop time** fields. The default start time is 0.0 seconds and the default stop time is 10.0 seconds.

Simulation time and actual clock time are not the same. For example, running a simulation for 10 seconds will usually not take 10 seconds. The amount of time it takes to run a simulation depends on many factors, including the model's complexity, the solver's step sizes, and the computer's clock speed.

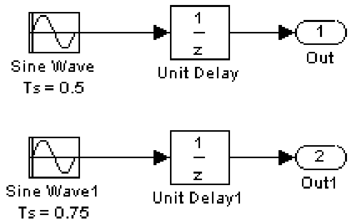
Solvers

Simulation of Simulink models involves the numerical integration of sets of ordinary differential equations (ODEs). Simulink provides a number of solvers for the simulation of such equations. Because of the diversity of dynamic system behavior, some solvers may be more efficient than others at solving a particular problem. To obtain accurate and fast results, take care when choosing the solver and setting parameters.

You can choose between variable-step and fixed-step solvers. *Variable-step solvers* can modify their step sizes during the simulation. They provide error control and zero crossing detection. *Fixed-step solvers* take the same step size during the simulation. They provide no error control and do not locate zero crossings. For a thorough discussion of solvers, see *Using MATLAB*.

Default solvers. If you do not choose a solver, Simulink chooses one based on whether your model has states:

- If the model has continuous states, `ode45` is used. `ode45` is an excellent general purpose solver. However, if you know that your system is stiff and if `ode45` is not providing acceptable results, try `ode15s`. For a definition of stiff, see the note on the next page.
- If the model has no continuous states, Simulink uses the variable-step solver called `discrete` and displays a message indicating that it is not using `ode45`. Simulink also provides a fixed-step solver called `discrete`. This model shows the difference between the two `discrete` solvers.



With sample times of 0.5 and 0.75, the *fundamental sample time* for the model is 0.25 seconds. The difference between the variable-step and the fixed-step `discrete` solvers is the time vector that each generates:

The fixed-step `discrete` solver generates this time vector:

```
[0.0 0.25 0.5 0.75 1.0 1.25 ...]
```

The variable-step `discrete` solver generates this time vector:

```
[0.0 0.5 0.75 1.0 1.5 2.0 2.25 ...]
```

The step size of the fixed-step `discrete` solver is the fundamental sample time. The variable-step `discrete` solver takes the largest possible steps.

Variable-step solvers. You can choose these variable-step solvers: `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, and `discrete`. The default is `ode45` for systems with states, or `discrete` for systems with no states.

- `ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver; that is, in computing $y(t_n)$, it needs only the

solution at the immediately preceding time point, $y(t_{n-1})$. In general, ode45 is the best solver to apply as a “first try” for most problems.

- ode23 is also based on an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than ode45 at crude tolerances and in the presence of mild stiffness. ode23 is a one-step solver.
- ode113 is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than ode45 at stringent tolerances. ode113 is a *multistep* solver; that is, it normally needs the solutions at several preceding time points to compute the current solution.
- ode15s is a variable order solver based on the numerical differentiation formulas (NDFs). These are related to but are more efficient than the backward differentiation formulas, BDFs (also known as Gear’s method). Like ode113, ode15s is a multistep method solver. If you suspect that a problem is stiff or if ode45 failed or was very inefficient, try ode15s.
- ode23s is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than ode15s at crude tolerances. It can solve some kinds of stiff problems for which ode15s is not effective.
- discrete (variable-step) is the solver Simulink chooses when it detects that your model has no continuous states.

NOTE For a *stiff* problem, solutions can change on a time scale that is very short compared to the interval of integration, but the solution of interest changes on a much longer time scale. Methods not designed for stiff problems are ineffective on intervals where the solution changes slowly because they use time steps small enough to resolve the fastest possible change. Jacobian matrices are generated numerically for ode15s and ode23s. For more information, see Shampine, L. F., *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, 1994.

Fixed-step solvers. You can choose these fixed-step solvers: ode5, ode4, ode3, ode2, ode1, and discrete.

- ode5 is the fixed-step version of ode45, the Dormand-Prince formula.
- ode4 is RK4, the fourth-order Runge-Kutta formula.

- `ode3` is the fixed-step version of `ode23`, the Bogacki-Shampine formula.
- `ode2` is Heun's method, also known as the improved Euler formula.
- `ode1` is Euler's method.
- `discrete` (fixed-step) is a fixed-step solver that performs no integration. It is suitable for models having no states and for which zero crossing detection and error control are not important.

If you think your simulation may be providing unsatisfactory results, see "Improving Simulation Performance and Accuracy" on page 4–19.

Solver Options

The default solver parameters provide accurate and efficient results for most problems. In some cases, however, tuning the parameters can improve performance (for more information about tuning these parameters, see "Improving Simulation Performance and Accuracy" on page 4–19). You can tune the selected solver by changing parameter values on the **Solver** page.

Step Sizes

For variable-step solvers, you can set the maximum and suggested initial step size parameters. By default, these parameters are automatically determined, indicated by the value `auto`.

For fixed-step solvers, you can set the fixed step size. The default is also `auto`.

Maximum step size. The **Max step size** parameter controls the largest time step the solver can take. The default is determined from the start and stop times:

$$h_{max} = \frac{t_{stop} - t_{start}}{50}$$

Generally, the default maximum step size is sufficient. If you are concerned about the solver missing significant behavior, change the parameter to prevent the solver from taking too large a step. If the time span of the simulation is very long, the default step size may be too large for the solver to find the solution. Also, if your model contains periodic or nearly periodic behavior and you know the period, set the maximum step size to some fraction (such as 1/4) of that period.

In general, for more output points, change the refine factor, not the maximum step size. For more information, see "Refine Factor" on page 4–12.

Initial step size. By default, the solvers select an initial step size by examining the derivatives of the states at the start time. If the first step size is too large, the solver may step over important behavior. The initial step size parameter is a *suggested* first step size. The solver tries this step size but reduces it if error criteria are not satisfied.

Error Tolerances

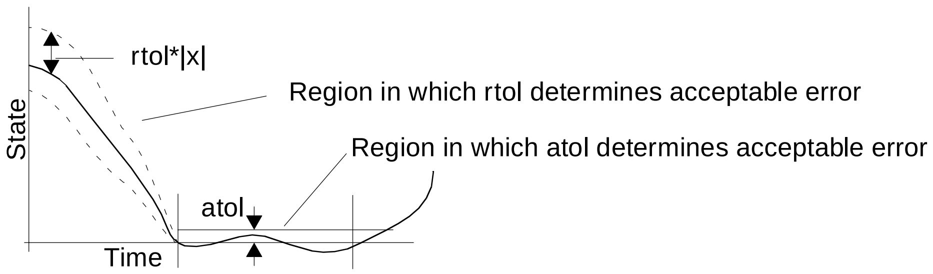
The solvers use standard local error control techniques to monitor the error at each time step. During each time step, the solvers compute the state values at the end of the step and also determine the *local error*, the estimated error of these state values. They then compare the local error to the *acceptable error*, which is a function of the relative tolerance (*rtol*) and absolute tolerance (*atol*). If the error is greater than the acceptable error for *any* state, the solver reduces the step size and tries again.

- *Relative tolerance* measures the error relative to the size of each state. The relative tolerance represents a percentage of the state's value. The default, 1e-3, means that the computed state will be accurate to within 0.1%.
- *Absolute tolerance* is a threshold error value. This tolerance represents the acceptable error as the value of the measured state approaches zero.

The error for the `i`th state, e_i, is required to satisfy:

$$e_i \leq max(rtol \times |x_i|, atol_i)$$

The figure below shows a plot of a state and the regions in which the acceptable error is determined by the relative tolerance and the absolute tolerance:



You might have to run a simulation more than once to determine an appropriate value for the absolute tolerance. If the magnitudes of the states vary widely, it might be appropriate to specify different absolute tolerance values for different states. You can do this on the Integrator block's dialog box.

The Maximum Order for ode15s

The `ode15s` solver is based on NDF formulas of order one through five. Although the higher order formulas are more accurate, they are less stable. If your model is stiff and requires more stability, reduce the maximum order to 2 (the highest order for which the NDF formula is A-stable). When you choose the `ode15s` solver, the dialog box displays this parameter.

As an alternative, you might try using the `ode23s` solver, which is a fixed-step, lower order (and A-stable) solver.

Output Options

The **Output options** area of the dialog box enables you to control how much output the simulation generates. You can choose from three popup options:

- Refine output
- Produce additional output
- Produce specified output only

Refine Factor. The **Refine output** choice provides additional output points when the simulation output is too coarse. This parameter provides an integer number of output points between time steps; for example, a refine factor of 2 provides output midway between the time steps, as well as at the steps. The default refine factor is 1.

To get smoother output, it is much faster to change the refine factor instead of reducing the step size. When the refine factor is changed, the solvers generate additional points by evaluating a continuous extension formula at those points. Changing the refine factor does not change the steps used by the solver.

The refine factor applies to variable-step solvers and is most useful when using `ode45`. The `ode45` solver is capable of taking large steps; when graphing simulation output, you may find that output from this solver is not sufficiently smooth. If this is the case, run the simulation again with a larger refine factor. A value of 4 should provide much smoother results.

Produce Additional Output. The **Produce additional output** choice enables you to specify directly those additional times at which the solver generates output. Similar to the refine factor, selecting this option does not change the step size. The additional output is produced using a continuous extension formula at those points.

Produce Specified Output Only. The **Produce specified output only** choice provides simulation output *only* at the specified output times. Specifying this parameter does not affect the step sizes used by the solver. This choice is useful when comparing different simulations to ensure that the simulations produce output at the same times.

Comparing Output Options. A sample simulation generates output at these times:

```
0, 2.5, 5, 8.5, 10
```

Choosing **Refine Output** and specifying a refine factor of 2 generates output at these times:

```
0, 1.25, 2.5, 3.75, 5, 6.75, 8.5, 9.25, 10
```

Choosing the **Produce Additional Output** option and specifying `[0:10]` generates output at these times:

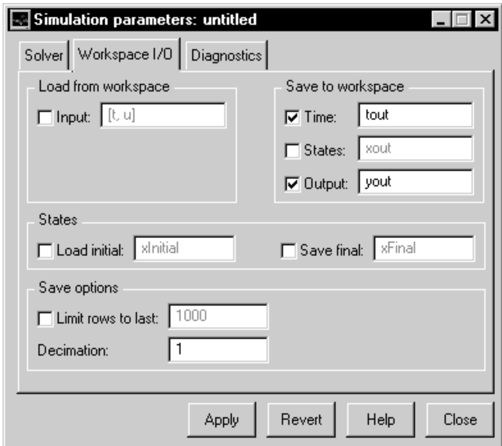
```
0, 1, 2, 2.5, 3, 4, 5, 6, 7, 8, 8.5, 9, 10
```

Choosing the **Produce Specified Output Only** option and specifying `[0:10]` generates output at these times:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

The Workspace I/O Page

You can direct simulation output to workspace variables and get input and initial states from the workspace. On the **Simulation Parameters** dialog box, select the **Workspace I/O** tab. This page appears:



Loading Input from the Base Workspace

Inport blocks in the top level of the model provide the means of supplying inputs from outside the system. These external inputs can be specified either as a MATLAB command expressed in terms of simulation time (t) or as a matrix that provides input values for all the Inport blocks. You specify external input by selecting the **Input** check box in the **Load from workspace** area of this dialog box page.

- You provide external inputs using a MATLAB command by specifying the command as a string. At each time point in the simulation, MATLAB evaluates the string; the result is the input. You can specify multiple external inputs by separating the expressions with a comma. For example:

`'sin(t)'`	inputs *sin(t)* at each time step
`'sin(t), cos(t)'`	provides two external inputs at each time step

- You can provide a matrix of time and input values. The first column must be a vector of times in ascending order. The remaining columns are interpreted as input values, where each column represents the input for a different Inport block (in sequential order) and each row is the input value for the

corresponding time point. When simulation times do not match the time values included in the time vector, Simulink linearly interpolates for the time and input values. This is the same method used for the From Workspace block, described in Chapter 9.

For example, these statements define a matrix that can be used to provide input values for three Inport blocks:

```
t = 0:0.1:10;
u = [cos(t), sin(t), tan(t)];
ut = [t, u];
```

Saving Output to the Workspace

You can specify return variables by selecting the **Time**, **States**, and/or **Output** check boxes in the **Save to workspace** area of this dialog box page. Specifying return variables causes Simulink to write values for the time, state, and output trajectories (as many as are selected) into the workspace.

To assign values to different variables, specify those variable names in the field to the right of the check boxes. To write output to more than one variable, specify the variable names in a comma-separated list.

The **Save options** area enables you to restrict the amount of output saved. To set a limit on the number of rows of data saved, select the check box labeled **Limit rows to last** and specify the number of rows to save. To apply a decimation factor, enter a value in the field to the right of the **Decimation** label. For example, a value of 2 saves every other point generated.

Loading and Saving States

Initial conditions, which are applied to the system at the start of the simulation, are generally set in the blocks. You can override initial conditions set in the blocks by specifying them in the **States** area of this page.

You can also save the final states for a simulation and apply them to another simulation. This feature might be useful when you want to save a steady-state solution and restart the simulation at that known state.

You load states by selecting the **Load initial** check box and specifying a state vector in the adjacent field. If the check box is not selected or the state vector is empty ([]), Simulink uses the initial conditions defined in the blocks.

You save the final states (the values of the states at the termination of the simulation) by selecting the **Save final** check box and entering a variable in the adjacent edit field.

When the Model Has Multiple States. If you want to specify the initial conditions for a model that has multiple states, you need to determine the order of the states. You can determine a model's initial conditions and the ordering of its states with this command:

```
[sizes, x0, xstord] = sys([], [], [], 0)
```

where `sys` is the model name. The command returns:

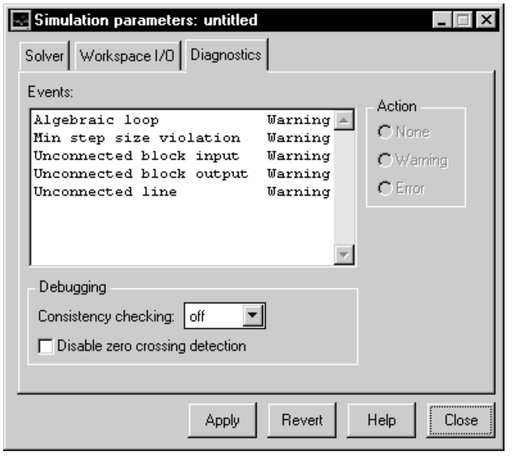
- `sizes`, a vector that indicates certain model characteristics. Only the first two elements apply to initial conditions: `sizes(1)` is the number of continuous states, and `sizes(2)` is the number of discrete states. The `sizes` vector is described in more detail in "Creating General Purpose S-Function Blocks" on page 8–42.
- `x0`, the block initial conditions.
- `xstord`, a string matrix that contains the full path name of all blocks in the model that have states. The order of the blocks in the `xstord` and `x0` vectors are the same.

For example, this statement obtains the values of the initial conditions and the ordering of the states for the `vdp` model (the example shows only the values for `sizes(1)`, the number of continuous states, and `sizes(2)`, the number of discrete states):

```
[sizes, x0, xstord] = vdp([], [], [], 0)

sizes =
   2
   0

x0 =
   2
   0

xstord =
   'vdp/Integrator1'
   'vdp/Integrator2'
```

The Diagnostics Page

You can indicate the desired action for many types of events or conditions that can be encountered during a simulation by selecting the **Diagnostics** tab on the **Simulation Parameters** dialog box. This dialog box appears:



For each event type, you can choose whether you want no message, a warning message, or an error message. A warning message does not terminate a simulation, but an error message does.

Consistency Checking

Consistency checking is a debugging tool that validates certain assumptions made by Simulink's ODE solvers. Its main use is to make sure that S-functions adhere to the same rules as Simulink built-in blocks. Because consistency checking results in a significant decrease in performance (up to 40%), it should generally be set to `off`. Use consistency checking to validate your S-functions and to help you determine the cause of unexpected simulation results.

To perform efficient integration, Simulink saves (caches) certain values from one time step for use in the next time step. For example, the derivatives at the end of a time step can generally be re-used at the start of the next time step. The solvers take advantage of this to avoid redundant derivative calculations.

Another purpose of consistency checking is to ensure that blocks produce constant output when called with a given value of *t* (time). This is important for the stiff solvers (`ode23s` and `ode15s`) because, while calculating the

Jacobian, the block's output functions may be called many times at the same value of *t*.

When consistency checking is enabled, Simulink recomputes the appropriate values and compares them to the cached values. If the values are not the same, a consistency error occurs. Simulink compares computed values for these quantities:

- Outputs
- Zero crossings
- Derivatives
- States

Disabling Zero Crossing Detection

You can disable zero crossing detection for a simulation. For a model that has zero crossings, disabling the detection of zero crossings may speed up the simulation but might have an adverse effect on the accuracy of simulation results.

This option disables zero crossing detection for those blocks that have intrinsic zero crossing detection. It does not disable zero crossing detection for the Hit Crossing block.

Improving Simulation Performance and Accuracy

Simulation performance and accuracy can be affected by many things, including the model design and choice of simulation parameters.

The solvers handle most model simulations accurately and efficiently with their default parameter values. However, some models will yield better results if you adjust solver and simulation parameters. Also, if you know information about your model's behavior, your simulation results can be improved if you provide this information to the solver.

Speeding Up the Simulation

Slow simulation speed can have many causes. Here are a few:

- Your model includes a MATLAB Fcn block. When a model includes a MATLAB Fcn block, the MATLAB interpreter is called at each time step, drastically slowing down the simulation. Use the built-in Fcn block or Elementary Math block whenever possible.
- Your model includes an M-file S-function. M-file S-functions also cause the MATLAB interpreter to be called at each time step. Consider either converting the S-function to a subsystem or to a C-MEX file S-function.
- Your model includes a Memory block. Using a Memory block causes the variable-order solvers (`ode15s` and `ode113`) to be reset back to order 1 at each time step.
- The maximum step size is too small. If you changed the maximum step size, try running the simulation again with the default value (`auto`).
- Did you ask for too much accuracy? The default relative tolerance (0.1% accuracy) is usually sufficient. For models with states that go to zero, if the absolute tolerance parameter is too small, the simulation may take too many steps around the near-zero state values. See the discussion of error in "Error Tolerances" on page 4–11.
- The time scale may be too long. Reduce the time interval.
- The problem may be stiff but you're using a nonstiff solver. Try using `ode15s`.
- The model uses sample times that are not multiples of each other. Mixing sample times that are not multiples of each other causes the solver to take small enough steps to ensure sample time hits for all sample times.

- The model contains an algebraic loop. The solutions to algebraic loops are iteratively computed at every time step. Therefore, they severely degrade performance. For more information, see “Algebraic Loops” in Chapter 10.
- Your model feeds a Random Number block into an Integrator. For continuous systems, use the Band-Limited White Noise block in the Sources library.

Improving Simulation Accuracy

To check your simulation accuracy, run the simulation over a reasonable time span. Then, reduce either the relative tolerance to 1e-4 (the default is 1e-3) or the absolute tolerance and run it again. Compare the results of both simulations. If the results are not significantly different, you can feel confident that the solution has converged.

If the simulation misses significant behavior at its start, reduce the initial step size to ensure that the simulation does not “step over” the significant behavior.

If the simulation results become unstable over time:

- Your system may be unstable.
- If you are using `ode15s`, you may need to restrict the maximum order to 2 (the maximum order for which the solver is A-stable) or try using the `ode23s` solver.

If the simulation results do not appear to be accurate:

- For a model that has states whose values approach zero, if the absolute tolerance parameter is too large, the simulation will take too few steps around areas of near-zero state values. Reduce this parameter value or adjust it for individual states in the Integrator dialog box.
- If reducing the absolute tolerances do not sufficiently improve the accuracy, reduce the size of the relative tolerance parameter to reduce the acceptable error and force smaller step sizes and more steps.

Running a Simulation from the Command Line

Entering simulation commands in the MATLAB command window or from an M-file enables you to run unattended simulations. You can perform Monte Carlo analysis by changing the parameters randomly and executing simulations in a loop. You can run a simulation from the command line using the `sim` command or the `set_param` command. Both are described below.

Using the sim Command

The full syntax of the command that runs the simulation is:

```
[t,x,y] = sim(model, timespan, options, ut, p1, ..., pn);
```

Only the `model` parameter is required. Parameters not supplied on the command are taken from the **Simulation Parameters** dialog box settings.

For detailed syntax for the `sim` command, see page 4–22. The `options` parameter is a structure that supplies additional simulation parameters, including the solver name and error tolerances. You define parameters in the `options` structure using the `simset` command, described on page 4–24. The simulation parameters are discussed earlier in this chapter.

Using the set_param Command

You can use the `set_param` command to start, stop, pause, or continue a simulation, or update a block diagram. Similarly, you can use the `get_param` command to check the status of a simulation. The format of the `set_param` command for this use is:

```
set_param('sys', 'SimulationCommand', 'cmd')
```

where `'sys'` is the name of the system and `'cmd'` is `'start'`, `'stop'`, `'pause'`, `'continue'`, or `'update'`.

The format of the `get_param` command for this use is:

```
get_param('sys', 'SimulationStatus')
```

Simulink returns `'stopped'`, `'initializing'`, `'running'`, `'paused'`, `'terminating'`, and `'external'` (used with Real-Time Workshop).

Purpose Simulate a Simulink model.

Syntax

```
[t,x,y] = sim(model,timespan,options,ut,p1,...,pn);
[t,x,y1, y2, ..., yn] = sim(model,timespan,options,ut,p1,...,pn);
```

Description The `sim` command simulates the specified Simulink model, integrating the system of ordinary differential equations that describe the model.

For block diagram models, only the `model` parameter is required. Values for all parameters specified as an empty matrix ([]) are taken from the simulation parameter settings. Any optional parameters specified in the command override the settings associated with the model.

For M-file and MEX-file S-functions, the `model` and `timespan` parameters are required. For models with continuous states, the `solver` parameter must be specified (using the `simset` command, described on page 4–24). For models with no continuous states, the solver defaults to `VariableStepDiscrete`.

Arguments

`t`	Returned time vector.
`x`	Returned state matrix, containing continuous states followed by discrete states.
`y`	Returned output matrix. For block diagram models, each column contains the output from a root-level Outport block, in port number order. If any Outport block has a vector input, its output takes the appropriate number of columns.
`y1,...,yn`	Can only be specified for block diagram models. `n` is the number of root-level Outport blocks. Each output is returned in the corresponding y_i.
`model`	Name of a block diagram, or MEX-file or M-file S-function.
`timespan`	Simulation start and stop time. Specify as one of these: `tFinal` to specify the stop time. The start time is 0. `[tStart tFinal]` to specify the start and stop times. `[tStart OutputTimes tFinal]` to specify the start and stop times and time points to be returned in `t`. Generally, `t` will include more time points. `OutputTimes` is equivalent to choosing **Produce additional output** on the dialog box.

`options`	Optional simulation parameters specified as a structure created by the `simset` command. See page 4–24.
`ut`	Optional external inputs to top-level Inport blocks. `ut` can be either one or more MATLAB commands expressed as a string or a matrix of values: If a *string* containing a function *u(t)*, MATLAB evaluates the function at each time step. If a *matrix* specified as `ut=[t,u1,...,u2]` where `t=[t1,...,tn]'`, the first column must be a vector of times in ascending order. The remaining columns are the corresponding values for the inputs. Simulink linearly interpolates between values when necessary.
`p1,...,pn`	Additional S-function parameters, specified as though this call were used: `sys = sfun(t, x, u, flag, p1, p2, ..., pn)` S-function parameters are used only in M-file and MEX-file systems.

Examples

This command simulates the `vdp` model using all default parameters:

```
[t,x,y] = sim('vdp')
```

This command simulates the `vdp` model using the parameter values associated with the model, but defines a value for the `Refine` parameter:

```
[t,x,y] = sim('vdp', [], simset('Refine',2));
```

This command simulates the `vdp` model for 1000 seconds, saving the last 100 rows of the return variables. The simulation outputs values for t and y only, but saves the final state vector in a variable called `xFinal`.

```
[t,x,y] = sim('vdp', 1000, simset('MaxRows', 100,
          'OutputVariables', 'ty', 'FinalStateName', 'xFinal'));
```

See Also

`simset`, `simget`

Purpose

Create or edit simulation parameters and solver properties for the `sim` command.

Syntax

```
options = simset(property, value, ...);
options = simset(old_opstruct, property, value, ...);
options = simset(old_opstruct, new_opstruct);
simset
```

Description

The `simset` command creates a structure called `options`, in which the named simulation parameters and solver properties have the specified values. All unspecified parameters and properties take their default values. It is only necessary to enter enough leading characters to uniquely identify the parameter or property. Case is ignored for parameters and properties.

`options = simset(property, value, ...)` sets the values of the named properties and stores the structure in `options`.

`options = simset(old_opstruct, property, value, ...)` modifies the named properties in `old_opstruct`, an existing structure.

`options = simset(old_opstruct, new_opstruct)` combines two existing options structures, `old_opstruct` and `new_opstruct`, into `options`. Any properties defined in `new_opstruct` overwrite the same properties defined in `old_opstruct`.

`simset` with no input arguments displays all property names and their possible values.

You cannot obtain or set values of these properties and parameters using the `get_param` and `set_param` commands.

Parameters

`AbsTol` `positive scalar {1e-6}`

Absolute error tolerance. This scalar applies to all elements of the state vector. `AbsTol` applies only to the variable-step solvers.

`Decimation` `positive integer {1}`

Decimation for output variables. Decimation factor applied to the return variables `t`, `x`, and `y`. A decimation factor of 1 returns every data logging time point, a decimation factor of 2 returns every other data logging time point, etc.

`DstWorkspace        base | {current} | parent`

Where to assign variables. This property specifies the workspace in which to assign any variables defined as return variables or as output variables on the To Workspace block.

`FinalStateName      string {''}`

Name of final states variable. This property specifies the name of a variable into which Simulink saves the model's states at the end of the simulation.

`FixedStep           positive scalar`

Fixed step size. This property applies only to the fixed-step solvers. If the model contains discrete components, the default is the fundamental sample time; otherwise, the default is one-fiftieth of the simulation interval.

`InitialState        vector {[]}`

Initial continuous and discrete states. The initial state vector consists of the continuous states (if any) followed by the discrete states (if any). `InitialState` supersedes the initial states specified in the model. The default, an empty matrix, causes the initial state values specified in the model to be used.

`InitialStep         positive scalar {auto}`

Suggested initial step size. This property applies only to the variable-step solvers. The solvers try a step size of `InitialStep` first. By default, the solvers determine an initial step size automatically.

`MaxOrder            1 | 2 | 3 | 4 | {5}`

Maximum order of `ode15s`. This property applies only to `ode15s`.

`MaxRows             nonnegative integer {0}`

Limit number of output rows. This property limits the number of rows returned in `t`, `x`, and `y` to the last `MaxRows` data logging time points. If specified as 0, the default, no limit is imposed.

`MaxStep             positive scalar {auto}`

Upper bound on the step size. This property applies only to the variable-step solvers and defaults to one-fiftieth of the simulation interval.

`OutputPoints`   `{specified} | all`

Determine output points. When set to `specified`, the solver produces outputs `t`, `x`, and `y` only at the times specified in `timespan`. When set to `all`, `t`, `x`, and `y` also include the time steps taken by the solver.

`OutputVariables`   `{txy} | tx | ty | xy | t | x | y`

Set output variables. If `'t'`, `'x'`, or `'y'` is missing from the property string, the solver produces an empty matrix in the corresponding output `t`, `x`, or `y`.

`Refine`   `positive integer {1}`

Output refine factor. This property increases the number of output points by the specified factor, producing smoother output. `Refine` applies only to the variable-step solvers. It is ignored if output times are specified.

`RelTol`   `positive scalar {1e-3}`

Relative error tolerance. This property applies to all elements of the state vector. The estimated error in each integration step satisfies

```
e(i) <= max(RelTol*abs(x(i)),AbsTol(i))
```

This property applies only to the variable-step solvers and defaults to 1e-3, which corresponds to accuracy within 0.1%.

```
Solver              VariableStepDiscrete |
                    ode45 | ode23 | ode113 | ode15s | ode23s |
                    FixedStepDiscrete |
                    ode5 | ode4 | ode3 | ode2 | ode1
```

Method to advance time. This property specifies which solver is used to advance time.

`SrcWorkspace`   `{base} | current | parent`

Where to evaluate expressions. This property specifies the workspace in which to evaluate MATLAB expressions defined in the model.

`Trace` `'minstep', 'siminfo', 'compile' {''}`

Tracing facilities. This property enables simulation tracing facilities (specify one or more as a comma-separated list):

- The `'minstep'` trace flag specifies that simulation will stop when the solution changes so abruptly that the variable-step solvers cannot take a step and satisfy the error tolerances. By default, Simulink issues a warning message and continues the simulation.
- The `'siminfo'` trace flag provides a short summary of the simulation parameters in effect at the start of simulation.
- The `'compile'` trace flag displays the compilation phases of a block diagram model.

`ZeroCross` `{on} | off`

Enable/disable location of zero crossings. This property applies only to the variable-step solvers. If set to `off`, variable-step solvers will not detect zero crossings for blocks having intrinsic zero crossing detection. The solvers adjust their step sizes only to satisfy error tolerance.

Examples

This command creates an options structure called `myopts` that defines values for the `MaxRows` and `Refine` parameters, using default values for other parameters:

```
myopts = simset('MaxRows', 100, 'Refine', 2);
```

This command simulates the `vdp` model for 10 seconds and uses the parameters defined in `myopts`:

```
[t,x,y] = sim('vdp', 10, myopts);
```

See Also

`sim`, `simget`

simget

Purpose

Get options structure properties and parameters.

Syntax

```
struct = simget(model)
value = simget(model, property)
```

Description

The `simget` command gets simulation parameter and solver property values for the specified Simulink model. If a parameter or property is defined using a variable name, `simget` returns the variable's value, not its name. If the variable does not exist in the workspace, Simulink issues an error message.

`struct = simget(model)` returns the current options structure for the specified Simulink model. The options structure is defined using the `sim` and `simset` commands.

`value = simget(model, property)` extracts the value of the named simulation parameter or solver property from the model.

`value = simget(OptionStructure, property)` extracts the value of the named simulation parameter or solver property from `OptionStructure`, returning an empty matrix if the value is not specified in the structure. `property` can be a cell array containing the list of parameter and property names of interest. If a cell array is used, the output is also a cell array.

You need to enter only as many leading characters of a property name as are necessary to uniquely identify it. Case is ignored for property names.

Examples

This command retrieves the options structure for the `vdp` model:

```
options = simget('vdp');
```

This command retrieves the value of the `Refine` property for the `vdp` model:

```
refine = simget('vdp', 'Refine');
```

See Also

`sim`, `simset`

5

Analyzing Simulation Results

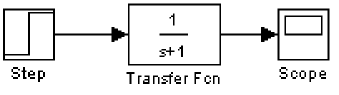
Viewing Output Trajectories

Output trajectories from Simulink can be plotted using one of three methods:

- Feeding a signal into either a Scope or an XY Graph block
- Writing output to return variables and using MATLAB plotting commands
- Writing output to the workspace using To Workspace blocks and plotting the results using MATLAB plotting commands

Using the Scope Block

You can use display output trajectories on a Scope block during a simulation. This simple model shows an example of the use of the Scope block:



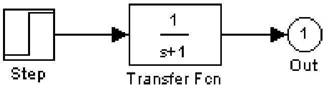
The display on the Scope shows the output trajectory. The Scope block enables you to zoom in on an area of interest or save the data to the workspace.

The XY Graph block enables you to plot one signal against another.

These blocks are described in Chapter 9.

Using Return Variables

By returning time and output histories, you can use MATLAB plotting commands to display and annotate the output trajectories.



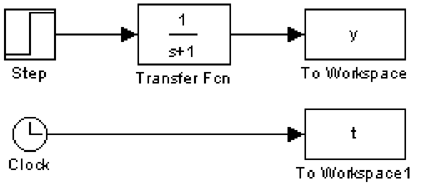
The block labeled Out is an Outport block from the Connections library. The output trajectory, yout, is returned by the integration solver. For more information, see Chapter 4.

You can also run this simulation from the **Simulation** menu by specifying variables for the time, output, and states on the **Workspace I/O** page of the **Simulation Parameters** dialog box. You can then plot these results using:

```
plot(tout,yout)
```

Using the To Workspace Block

The To Workspace block can be used to return output trajectories to the MATLAB workspace. The model below illustrates this use:



The variables `y` and `t` appear in the workspace when the simulation is complete. The time vector is stored by feeding a Clock block into a To Workspace block. The time vector can also be acquired by entering a variable name for the time on the **Workspace I/O** page of the **Simulation Parameters** dialog box for menu-driven simulations, or by returning it using the `sim` command (see Chapter 4 for more information).

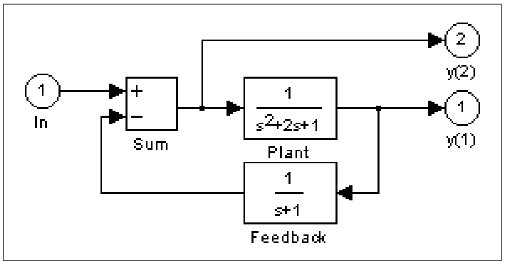
The To Workspace block can accept a vector input, with each input element's trajectory stored as a column vector in the resulting workspace variable.

Linearization

Simulink provides the `linmod` and `dlinmod` functions to extract linear models in the form of the state-space matrices *A*, *B*, *C*, and *D*. State-space matrices describe the linear input-output relationship as

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

where *x*, *u*, and *y* are state, input, and output vectors, respectively. For example, the following model is called `lmod`.



To extract the linear model of this Simulink system, enter this command:

```
[A,B,C,D] = linmod('lmod')
A =
   -2    -1    -1
    1     0     0
    0     1    -1
B =
    1
    0
    0
C =
    0     1     0
    0     0    -1
D =
    0
    1
```

Inputs and outputs must be defined using Inport and Outport blocks from the Connections library. Source and sink blocks do not act as inputs and outputs. Inport blocks can be used in conjunction with source blocks using a Sum block.

Once the data is in the state-space form or converted to an LTI object, you can apply functions in the Control System Toolbox for further analysis:

- Conversion to an LTI object:

```
sys = ss(A,B,C,D);
```

- Bode phase and magnitude frequency plot:

```
bode(A,B,C,D) or bode(sys)
```

- Linearized time response:

```
step(A,B,C,D) or step(sys)
impulse(A,B,C,D) or impulse(sys)
lsim(A,B,C,D,u,t) or lsim(sys,u,t)
```

Other functions in the Control System Toolbox and Robust Control Toolbox can be used for linear control system design.

When the model is nonlinear, an operating point may be chosen at which to extract the linearized model. The nonlinear model is also sensitive to the perturbation sizes at which the model is extracted. These must be selected to balance the trade-off between truncation and roundoff error. Extra arguments to `linmod` specify the operating point and perturbation points:

```
[A,B,C,D] = linmod('sys', x, u, pert, xpert, upert)
```

For discrete systems or mixed continuous and discrete systems, use the function `dlinmod` for linearization. This has the same calling syntax as `linmod` except that the second right-hand argument must contain a sample time at which to perform the linearization. For more information, see the description of the `linmod` command on page 5–9.

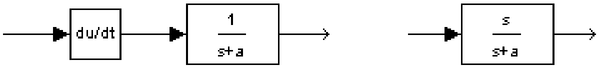
Using `linmod` to linearize a model that contains Derivative or Transport Delay blocks can be troublesome. Before linearizing, replace these blocks with specially designed blocks that avoid the problems. These blocks are in the Simulink Extras library in the Linearization sublibrary. You access the Extras library by opening the Blocksets & Toolboxes icon.

- For the Derivative block, use the Switched derivative for linearization.
- For the Transport Delay block, use the Switched transport delay for linearization. (Using this block requires that you have the Control System Toolbox.)

When using a Derivative block, you can also try to incorporate the derivative term in other blocks. For example, if you have a Derivative block in series with a Transfer Fcn block, it is better implemented (although this is not always possible) with a single Transfer Fcn block of the form

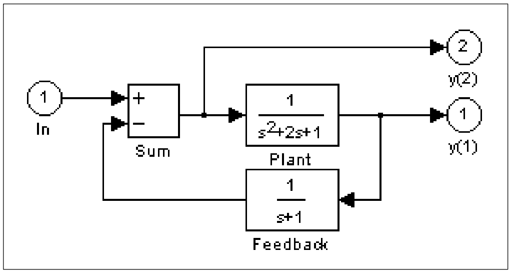
$$\frac{s}{s+a}$$

In this example, the blocks on the left of this figure can be replaced by the block on the right:



Equilibrium Point Determination (trim)

The Simulink `trim` function determines steady-state equilibrium points. Consider, for example, this model, called `lmod`:



You can use the `trim` function to find the values of the input and the states that set both outputs to 1. First, make initial guesses for the state variables (x) and input values (u), then set the desired value for the output (y):

```
x = [0; 0; 0];
u = 0;
y = [1; 1];
```

Use index variables to indicate which variables are fixed and which can vary:

```
ix = [];      % Don't fix any of the states
iu = [];      % Don't fix the input
iy = [1;2];   % Fix both output 1 and output 2
```

Invoking `trim` returns the solution. Your results may differ due to roundoff error.

```
[x,u,y,dx] = trim('lmod',x,u,y,ix,iu,iy)

x =
   0.0000
   1.0000
   1.0000
u =
   2
y =
   1.0000
   1.0000
```

```
dx =
   1.0e-015 *
    -0.2220
    -0.0227
     0.3331
```

Note that there may be no solution to equilibrium point problems. If that is the case, `trim` returns a solution that minimizes the maximum deviation from the desired result after first trying to set the derivatives to zero. For a description of the `trim` syntax, see page 5–13.

Purpose

Extract the linear state-space model of a system around an operating point.

Syntax

```
[A,B,C,D] = linfun('sys')
[A,B,C,D] = linfun('sys', x, u)
[A,B,C,D] = linfun('sys', x, u, pert)
[A,B,C,D] = linfun('sys', x, u, pert, xpert, upert)
```

Arguments

linfun	`linmod`, `dlinmod`, or `linmod2`.
`sys`	The name of the Simulink system from which the linear model is to be extracted.
`x` and `u`	The state and the input vectors. If specified, they set the operating point at which the linear model is to be extracted.
`pert`	Optional scalar perturbation factor used for both `x` and `u`. If not specified, a default value of 1e-5 is used.
`xpert` and `upert`	Optional vectors that explicitly set perturbation levels for individual states and inputs. If specified, the `pert` argument is ignored. The `i`th state `x` is perturbed to `x(i) + xpert(i)` The `j`th input `u` is perturbed to `u(j) + upert(j)`

Description

`linmod` obtains linear models from systems of ordinary differential equations described as Simulink models. `linmod` returns the linear model in state-space form, A, B, C, D, which describes the linearized input-output relationship:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

`[A,B,C,D] = linmod('sys')` obtains the linearized model of `sys` around an operating point with the state variables `x` and the input `u` set to zero.

`linmod` perturbs the states around the operating point to determine the rate of change in the state derivatives and outputs (Jacobians). This result is used to calculate the state-space matrices. Each state `x(i)` is perturbed to

$$x(i) + \Delta(i)$$

where

$$\Delta(i) = \delta(1 + |x(i)|)$$

Likewise the `j`th input is perturbed to

$$u(j) + \Delta(j)$$

where

$$\Delta(j) = \delta(1 + |u(j)|)$$

Discrete-Time System Linearization

The function `dlinmod` can linearize discrete, multirate, and hybrid continuous and discrete systems at any given sampling time. Use the same calling syntax for `dlinmod` as for `linmod`, but insert the sample time at which to perform the linearization as the second argument. For example:

```
[Ad,Bd,Cd,Dd] = dlinmod('sys', Ts, x, u);
```

produces a discrete state-space model at the sampling time `Ts` and the operating point given by the state vector `x` and input vector `u`. To obtain a continuous model approximation of a discrete system, set `Ts` to `0`.

For systems composed of linear, multirate, discrete, and continuous blocks, `dlinmod` produces linear models having identical frequency and time responses (for constant inputs) at the converted sampling time `Ts`, provided that:

- `Ts` is an integer multiple of all the sampling times in the system.
- `Ts` is not less than the slowest sample time in the system.
- The system is stable.

It is possible for valid linear models to be obtained when these conditions are not met.

Computing the eigenvalues of the linearized matrix `Ad` provides an indication of the stability of the system. The system is stable if `Ts>0` and the eigenvalues are within the unit circle, as determined by this statement:

```
all(abs(eig(Ad))) < 1
```

Likewise, the system is stable if `Ts = 0` and the eigenvalues are in the left half plane, as determined by this statement:

```
all(real(eig(Ad))) < 0
```

When the system is unstable and the sample time is not an integer multiple of the other sampling times, `dlinmod` produces `Ad` and `Bd` matrices, which may be complex. The eigenvalues of the `Ad` matrix in this case still, however, provide a good indication of stability.

You can use `dlinmod` to convert the sample times of a system to other values or to convert a linear discrete system to a continuous system or vice versa.

The frequency response of a continuous or discrete system can be found by using the `bode` command.

An Advanced Form of Linearization

The `linmod2` routine provides an advanced form of linearization. This routine takes longer to run than `linmod`, but may produce more accurate results.

The calling syntax for `linmod2` is similar to that used for `linmod`, but functions differently. For instance, `linmod2('sys',x,u)` produces a linear model as does `linmod`; however, the perturbation levels for each state-space matrix element are set individually to attempt to minimize roundoff and truncation errors.

`linmod2` tries to balance roundoff error (caused by small perturbation levels, which cause errors associated with finite precision mathematics) and truncation error (caused by large perturbation levels, which invalidate the piecewise linear approximation).

With the form `[A,B,C,D] = linmod2('sys',x,u,pert)`, the variable `pert` indicates the lowest level of perturbation that can be used; the default is 1e-8. `linmod2` has the advantage that it can detect discontinuities and produce warning messages, such as the following:

```
Warning:  discontinuity detected at A(2,3)
```

When such a warning occurs, try a different operating point at which to obtain the linear model.

With the form

```
[A,B,C,D] = linmod2('sys',x,u,pert,Apert,Bpert,Cpert,Dpert)
```

the variables `Apert`, `Bpert`, `Cpert`, and `Dpert` are matrices used to set the perturbation levels for each state and input combination; therefore, the `ij`th

element of Apert is the perturbation level associated with obtaining the ijth element of the A matrix. Return default perturbation sizes with

```
[A,B,C,D,Apert,Bpert,Cpert,Dpert] = linmod2('sys', x, u);
```

Notes

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable pert to a two-element vector, where the second element is used to set the value of t at which to obtain the linear model.

When the model being linearized is itself a linear model, the problem of truncation error no longer exists; therefore, you can set the perturbation levels to whatever value is desired. A relatively high value is generally preferable, since this tends to reduce roundoff error. The operating point used does not affect the linear model obtained.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a string variable that contains the block name associated with each state can be obtained using

```
[sizes,x0,xstring] = sys
```

where xstring is a vector of strings whose ith row is the block name associated with the ith state. Inputs and outputs are numbered sequentially on the diagram.

For single-input multi-output systems, you can convert to transfer function form using the routine ss2tf or to zero-pole form using ss2zp. You can also convert the linearized models to LTI objects using ss. This function produces an LTI object in state-space form that can be further converted to transfer function or zero-pole-gain form using tf or zpk.

Linearizing a model that contains Derivative or Transport Delay blocks can be troublesome. For more information, see “Linearization” on page 5–4.

Purpose Determine steady state parameters that satisfy input, output, and state conditions.

Syntax

```
[x,u,y,dx] = trim('sys')
[x,u,y,dx] = trim('sys',x0,u0,y0)
[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy)
[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx)
[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options)
[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options,t)
[x,u,y,dx,options] = trim('sys',...)
```

Description `trim` attempts to find values for the inputs `u` and states `x` that set the state derivatives to zero. Such a point is known as an equilibrium point, which occurs when the system is in steady state.

Since the problem is usually not unique, specific values of the state `x`, the input `u`, and the outputs `y` can often be fixed.

`[x,u,y] = trim('sys')` tries to find an equilibrium point such that the maximum absolute value of `[x–x0,u,y]` is minimized (`x0` is the initial state of the system). You can obtain `x0` using this command:

```
[sizes,x0,xstr] = sys([],[],[],0)
```

`[x,u,y] = trim('sys',x0,u0,y0)` specifies initial starting guesses for `x`, `u`, and `y`. Here, the maximum value of `abs([x–x0; u–u0; y–y0])` is minimized.

Individual elements of `x`, `u`, and `y` can be fixed using the syntax

```
trim('sys', x0, u0, y0, ix, iu, iy)
```

The integer vectors `ix`, `iu`, and `iy` single out elements in `x0`, `u0`, and `y0` to be fixed. Because there is no guarantee of a solution point, the method finds steady state values that minimize the maximum value of

```
abs([x(ix)–x0(ix); u(iu)–u0(iu); y(iy)–y0(iy)])
```

`trim` uses a constrained optimization method, which restricts the state derivatives to be zero, and solves a minimax problem formed from the desired values for `x`, `u`, and `y`. It is possible that no feasible solution exists to this problem, in which case `trim` minimizes the worst case deviation from zero of the state derivatives.

To fix the derivatives to nonzero values, use

```
[x,u,y,dx] = trim('sys', x0, u0, y0, ix, iu, iy, dx0, idx)
```

where dx0 represents the desired derivative values, and idx indexes the elements in dx to be fixed.

The trim command takes an optional argument, options, used with optimization routines. The options argument is described in detail in the documentation for the Optimization Toolbox.

Examples

Consider a linear state-space model

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

The A, B, C, and D matrices are as follows in a system called sys:

```
A = [-0.09 -0.01;  1   0];
B = [ 0    -7;     0  -2];
C = [ 0     2;     1  -5];
D = [-3     0;     1   0];
```

Example 1

To find an equilibrium point, use

```
[x,u,y,dx,options] = trim('sys')

x =
   0
   0
u =
   0
y =
   0
   0
dx =
   0
   0
```

The number of iterations taken is

```
options(10)
ans =
      7
```

Example 2

To find an equilibrium point near `x = [1;1], u = [1;1]` enter

```
x0 = [1;1];
u0 = [1;1];
[x,u,y,dx,options] = trim('sys', x0, u0);
```

```
x =
    1.0e-11 *
   -0.1167
   -0.1167
u =
    0.3333
    0.0000
y =
   -1.0000
    0.3333
dx =
    1.0e-11 *
    0.4214
    0.0003
```

The number of iterations taken is

```
options(10)
ans =
      25
```

Example 3

To find an equilibrium point with the outputs fixed to `1`, use

```
y = [1;1];
iy = [1;2];
[x,u,y,dx] = trim('sys', [], [], y, [], [], iy)
```

```
x =
    0.0009
   -0.3075
u =
   -0.5383
    0.0004
y =
    1.0000
    1.0000
```

```
dx =
    1.0e-16 *
   -0.0173
    0.2396
```

Example 4

To find an equilibrium point with the outputs fixed to 1 and the derivatives set to 0 and 1, use

```
y = [1;1];
iy = [1;2];
dx = [0;1];
idx = [1;2];
[x,u,y,dx,options] = trim('sys',[],[],y,[],[],iy,dx,idx)
```

```
x =
    0.9752
   -0.0827
u =
   -0.3884
   -0.0124
y =
    1.0000
    1.0000
dx =
    0.0000
    1.0000
```

The number of iterations taken is

```
options(10)
ans =
      13
```

Limitations

When a steady state solution is found, better values for `x`, `u`, and `y` may exist because there is no guarantee of global solutions unless the optimization problem is univariate (i.e., has a single minimum). Thus, it is important to try a number of starting guesses for `x`, `u`, and `y` if you seek global solutions. `trim` does not work well when the system has discontinuities.

Algorithm

`trim` uses a constrained optimization routine, which restricts the state derivatives to zero, and solves a minimax problem formed from the desired

values for x, u, and y. The underlying method is Sequential Quadratic Programming.

6

Using Masks to Customize Blocks

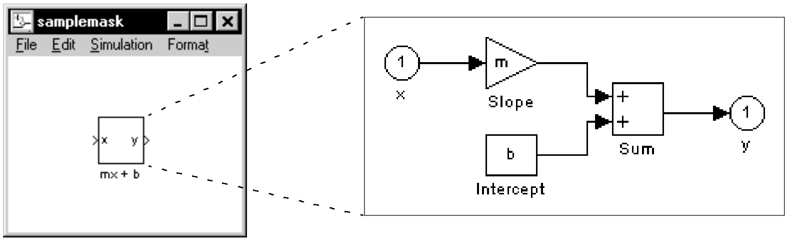
Introduction

Masking is a powerful Simulink feature that enables you to customize the dialog box and icon for a subsystem. With masking, you can:

- Simplify the use of your model by replacing many dialog boxes in a subsystem with a single one. Instead of requiring the user of the model to open each block and enter parameter values, those parameter values can be entered on the mask dialog box and passed to the blocks in the masked subsystem.
- Provide a more descriptive and helpful user interface by defining a dialog box with your own block description, parameter field labels, and help text.
- Define commands that compute variables whose values depend on block parameters.
- Create a block icon that depicts the subsystem's purpose.
- Prevent unintended modification of subsystems by hiding their contents behind a customized interface.

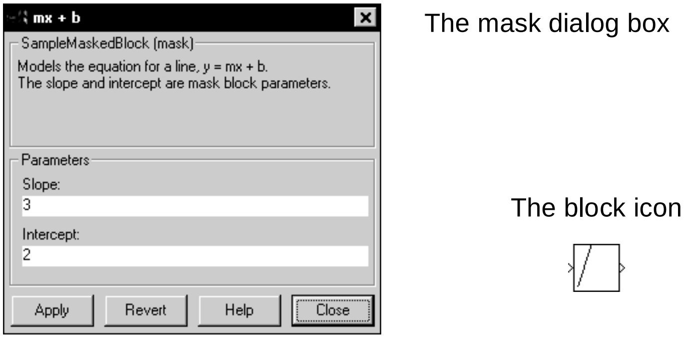
A Sample Masked Subsystem

This simple subsystem models the equation for a line, `y = mx + b`:



Ordinarily, when you double-click on a Subsystem block, the Subsystem block opens, displaying its blocks in a separate window. The `mx + b` subsystem contains a Gain block, named Slope, whose Gain parameter is specified as `m` and a Constant block, named Intercept, whose Constant value parameter is specified as `b`. These parameters represent the slope and intercept of a line.

In this example, a custom dialog box and icon are created for the subsystem. One dialog box contains prompts for both the slope and the intercept. After creating the mask, double-clicking on the Subsystem block opens the mask dialog box. The mask dialog box and icon look like this:



The user of this subsystem enters values for **Slope** and **Intercept** into the mask dialog box. Simulink makes these values available to all the blocks in the underlying subsystem. Masking this subsystem creates a self-contained functional unit with its own application-specific parameters, Slope and Intercept. The mask maps these *mask parameters* to the generic parameters of the underlying blocks. The complexity of the subsystem is encapsulated by a new interface that has the look and feel of a built-in Simulink block.

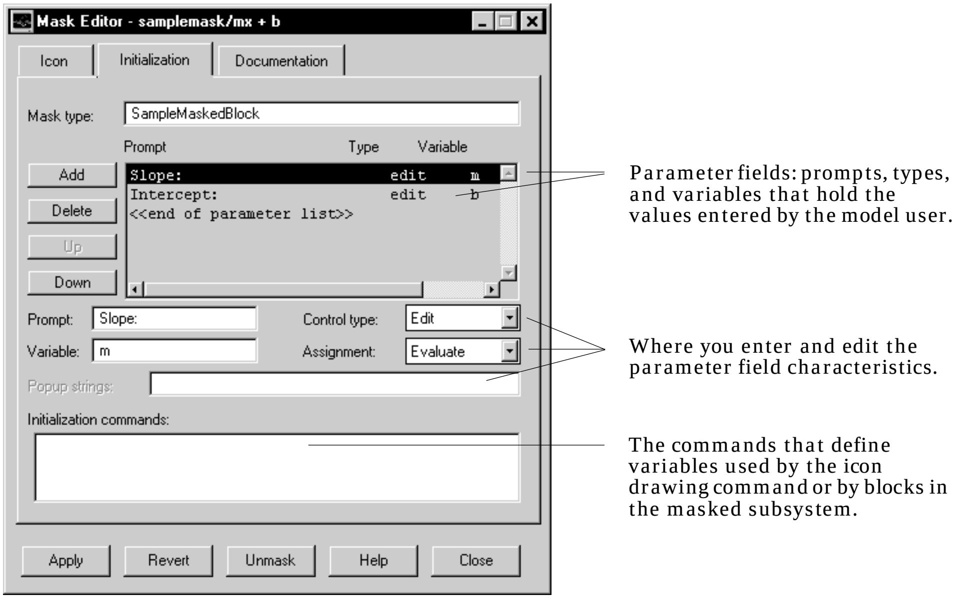
Creating a mask for this subsystem involves these tasks:

- Specifying the prompts for the mask dialog box parameters. In this example, the mask dialog box has prompts for the slope and intercept.
- Specifying the variable name used to store the value of each parameter.
- Entering the documentation of the block, consisting of the block description and the block help text.
- Specifying the drawing command that creates the block icon.
- Specifying the commands that provide the variables needed by the drawing command (there are none in this example).

Creating Mask Dialog Box Prompts

To create the mask for this subsystem, select the Subsystem block and choose **Mask Subsystem** from the **Edit** menu.

The mask dialog box shown above is created largely on the **Initialization** page of the Mask Editor. For this sample model, the page looks like this:

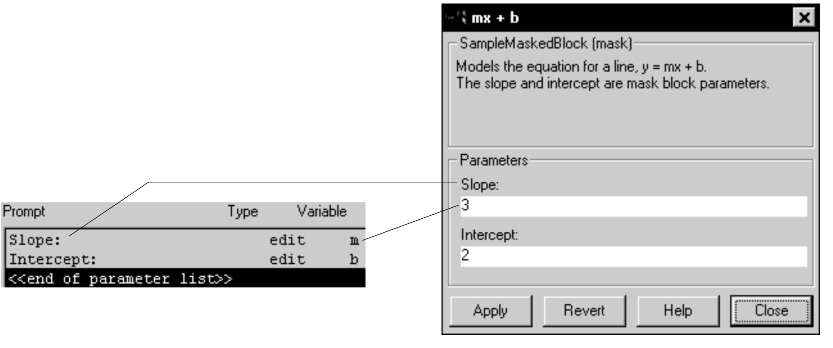


The Mask Editor enables you to specify these attributes of a mask parameter:

- The prompt – the text label that describes the parameter.
- The control type – the style of user interface control that determines how parameter values are entered or selected.
- The variable – the name of the variable that will store the parameter value.

Generally, it is convenient to refer to masked parameters by their prompts. In this example, the parameter associated with slope is referred to as the Slope parameter, and the parameter associated with intercept is referred to as the Intercept parameter.

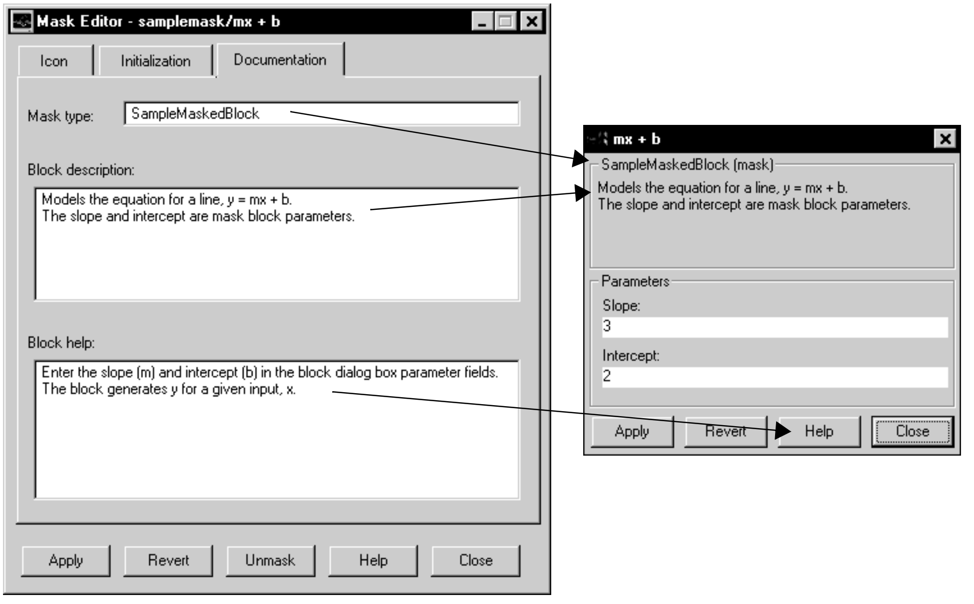
The slope and intercept are defined as edit controls. This means that the user types values into edit fields in the mask dialog box. These values are stored in variables in the *mask workspace*, described on page 6–14. Masked blocks can access variables only in the mask workspace. In this example, the value entered for the slope is assigned to the variable `m`. The Slope block in the masked subsystem gets the value for the slope parameter from the mask workspace. This figure shows how the slope parameter definitions in the Mask Editor map to the actual mask dialog box parameters.



After you have created the mask parameters for slope and intercept, press the **Close** button. Then, double-click on the Subsystem block to open the newly constructed dialog box. Enter 3 for the **Slope** and 2 for the **Intercept** parameters.

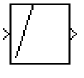
Creating the Block Description and Help Text

The mask type, block description, and help text are defined on the **Documentation** page. For this sample masked block, the page looks like this:

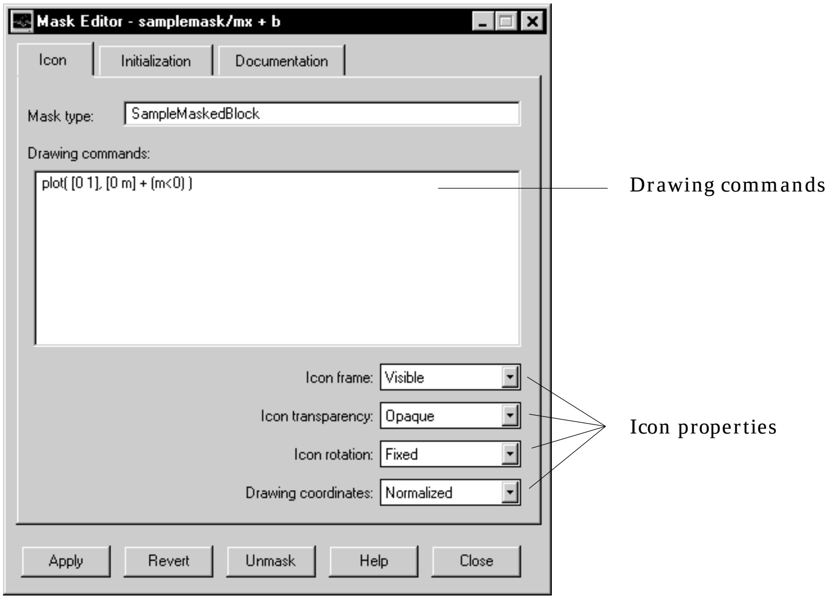


Creating the Block Icon

So far, a customized dialog box has been created for the `mx + b` subsystem. However, the Subsystem block still displays the generic Simulink subsystem icon. An appropriate icon for this masked block is a plot that indicates the slope of the line. For a slope of 3, that icon looks like this:



The block icon is defined on the **Icon** page. For this block, the **Icon** page looks like this:



The drawing command plots a line from `(0,0)` to `(0,m)`. If the slope is negative, the line is shifted up by 1 to keep it within the visible drawing area of the block.

The drawing commands have access to all of the variables in the mask workspace. As different values of slope are entered, the icon automatically updates the slope of the plotted line.

Selecting **Normalized** as the **Drawing coordinates** parameter, located at the bottom of the list of icon properties, specifies that the icon is drawn in a frame whose bottom-left corner is (0,0) and whose top-right corner is (1,1). This parameter is described later in this chapter.

Summary

This discussion of the steps involved in creating a sample mask introduced you to these tasks:

- Defining dialog box prompts and their characteristics
- Defining the masked block description and help text
- Defining the command that creates the masked block icon

The remainder of this chapter discusses the Mask Editor in more detail.

The Mask Editor (An Overview)

To mask a subsystem (you can only mask Subsystem blocks), select the Subsystem block, then choose **Mask Subsystem** from the **Edit** menu. The Mask Editor appears. The Mask Editor consists of three pages, each handling a different aspect of the mask:

- The **Initialization** page enables you to define and describe mask dialog box parameter prompts, name the variables associated with the parameters, and specify initialization commands.
- The **Icon** page enables you to define the block icon.
- The **Documentation** page enables you to define the mask type, and specify the block description and the block help.

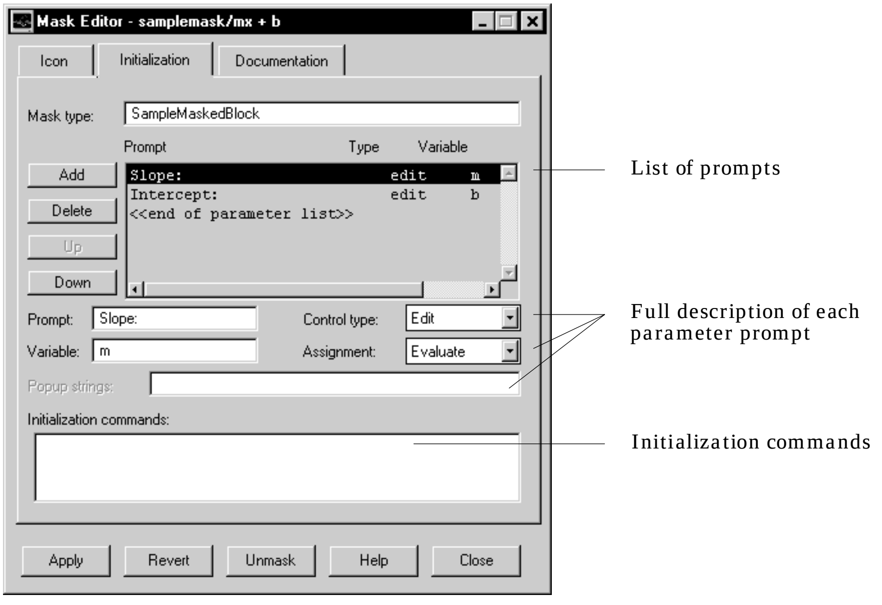
Five buttons appear along the bottom of the Mask Editor:

- The **Apply** button creates or changes the mask using the information that appears on all masking pages. The Mask Editor remains open.
- The **Revert** button refreshes the page, replacing all field values with the information that appeared when the Mask Editor was most recently opened. All mask parameters on all Mask Editor pages revert to their original values.
- The **Unmask** button deactivates the mask and closes the Mask Editor. The mask information is retained so that the mask can be reactivated. To reactivate the mask, select the block and choose **Create Mask**. The Mask Editor opens, displaying the previous settings. The inactive mask information is discarded when the model is closed and cannot be recovered.
- The **Help** button displays the contents of this chapter.
- The **Close** button applies the mask settings on all pages and closes the Mask Editor.

To see the system under the mask without unmasking it, select the Subsystem block, then choose **Look Under Mask** from the **Edit** menu. This command opens the subsystem. The block's mask is not affected.

The Initialization Page

The mask interface enables the user of the masked system to enter parameter values for blocks within the masked system. You create the mask interface by defining prompts for parameter values on the **Initialization** page. The **Initialization** page for the mx+b sample masked system looks like this:



Prompts and Associated Variables

A *prompt* provides information that helps the user enter or select a value for a block parameter. Prompts appear on the mask dialog box in the order they appear in the **Prompt** list.

When you define a prompt, you also specify the variable that is to store the parameter value, choose the style of control for the prompt, and indicate how the value is to be stored in the variable.

If the **Assignment** type is **Evaluate**, the value entered by the user is evaluated by MATLAB before it is assigned to the variable. If the type is **Literal**, the value entered by the user is not evaluated, but is assigned to the variable as a string.

For example, if the user enters the string `gain` in an edit field and the **Assignment** type is **Evaluate**, the string `gain` is evaluated by MATLAB and the result is assigned to the variable. If the type is **Literal**, the string is not evaluated by MATLAB so the variable contains the string `'gain'`.

If you need both the string entered as well as the evaluated value, choose **Literal**. Then use the MATLAB `eval` command in the initialization commands. For example, if `LitVal` is the string `'gain'`, then to obtain the evaluated value, use the command:

```
value = eval(LitVal)
```

In general, most parameters use an **Assignment** type of **Evaluate**.

Creating the First Prompt

To **create** the first prompt in the list, enter the prompt in the **Prompt** field, the variable that is to contain the parameter value in the **Variable** field, and choose a control style and an assignment type.

Inserting a Prompt

To **insert** a prompt in the list:

1. Select the prompt that appears immediately *below* where you want to insert the new prompt and click on the **Add** button to the left of the prompt list.
2. Enter the text for the prompt in the **Prompt** field. Enter the variable that is to hold the parameter value in the **Variable** field.

Editing a Prompt

To **edit** an existing prompt:

1. Select the prompt in the list. The prompt, variable name, control style, and assignment type appear in the fields below the list.
2. Edit the appropriate value. When you click the mouse outside the field or press the **Enter** or **Return** key, Simulink updates the prompt.

Deleting a Prompt

To **delete** a prompt from the list:

1. Select the prompt you want to delete.
2. Click on the **Delete** button to the left of the prompt list.

Moving a Prompt

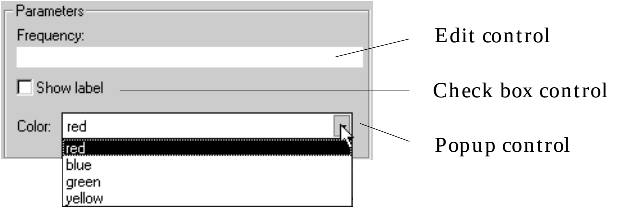
To **move** a prompt in the list:

1 Select the prompt you want to move.

2 To move the prompt up one position in the prompt list, click on the **Up** button to the left of the prompt list. To move the prompt down one position, click on the **Down** button.

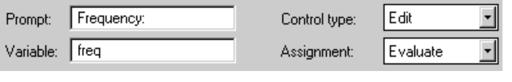
Control Types

Simulink enables you to choose how parameter values are entered or selected. You can create three styles of controls: edit fields, check boxes, and popups. For example, this figure shows the parameter area of a mask dialog box which uses all three styles of controls (and the popup is open):



Defining an Edit Control

An *edit field* enables the user to enter a parameter value by typing it into a field. This figure shows how the prompt for the sample edit control was defined:

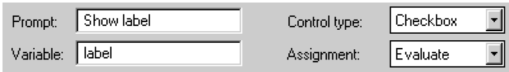


The value of the variable associated with the parameter (`freq`) is determined by the **Assignment** type defined for the prompt:

Assignment	Value
Evaluate	The result of evaluating the expression entered in the field.
Literal	The actual string entered in the field.

Defining a Check Box Control

A *check box* enables the user to choose between two alternatives by selecting or deselecting a check box. This figure shows how the sample check box control was defined:

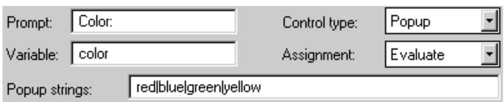


The value of the variable associated with the parameter (`label`) depends on whether the check box is selected and the **Assignment** type defined for the prompt:

Check box	Evaluated Value	Literal Value
Checked	1	`'on'`
Not checked	0	`'off'`

Defining a Popup Control

A *popup* enables the user to choose a parameter value from a list of possible values. You specify the list in the **Popup strings** field, separating items with a vertical line (|). This figure shows how the sample popup control was defined:



The value of the variable associated with the parameter (`color`) depends on the item selected from the popup list and the **Assignment** type defined for the prompt:

Assignment	Value
Evaluate	The index of the value selected from the list, starting with 1. For example, if the third item is selected, the parameter value is 3.
Literal	A string that is the value selected. If the third item is selected, the parameter value is `'green'`.

Default Values for Masked Block Parameters

To change default parameter values in a masked library block, follow these steps:

1. Unlock the library.
2. Open the block to access its dialog box, fill in the desired default values, and close the dialog box.
3. Save the library.

When the block is copied into a model and opened, the default values appear on the block's dialog box.

For more information about libraries, see Chapter 3.

Initialization Commands

Initialization commands define variables that reside in the mask workspace. These variables can be used by all initialization commands defined for the mask, by blocks in the masked subsystem, and by commands that draw the block icon (drawing commands).

Simulink executes the initialization commands when:

- The model is loaded.
- The simulation is started or the block diagram is updated.
- The masked block is rotated.
- The block's icon needs to be redrawn and the plot commands depend on variables defined in the initialization commands.

Initialization commands are valid MATLAB expressions, consisting of MATLAB functions, operators, and variables defined in the mask workspace. Initialization commands cannot access base workspace variables. Terminate initialization commands with a semi-colon to avoid echoing results to the command window.

The Mask Workspace

Simulink creates a local workspace, called a *mask workspace*, when:

- The mask contains initialization commands, or
- The mask defines prompts and associates variables with those prompts.

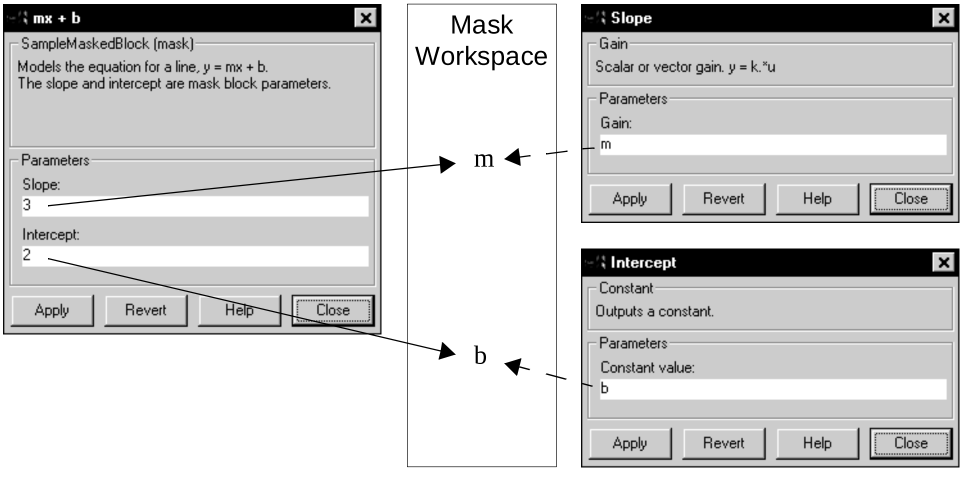
Masked blocks cannot access the base workspace or other mask workspaces.

The contents of a mask workspace include the variables associated with the mask's parameters and variables defined by initialization commands. The variables in the mask workspace can be accessed by the masked block. If the block is a subsystem, they can also be accessed by all blocks in the subsystem.

Mask workspaces are analogous to the local workspaces used by M-file functions. You can think of the expressions entered into the dialog boxes of the underlying blocks and the initialization commands entered on the Mask Editor as lines of an M-file function. Using this analogy, the local workspace for this "function" is the mask workspace.

In the `mx + b` example, described earlier in this chapter, the Mask Editor explicitly creates `m` and `b` in the mask workspace by associating a variable with a mask parameter. However, variables in the mask workspace are not explicitly assigned to blocks underneath the mask. Instead, blocks beneath the mask have access to all variables in the mask workspace. It may be instructive to think of the underlying blocks as "looking into" the mask workspace.

The figure below shows the mapping of values entered in the mask dialog box to variables in the mask workspace (indicated by the solid line) and the access of those variables by the underlying blocks (indicated by the dashed line):



Debugging Initialization Commands

You can debug initialization commands in these ways:

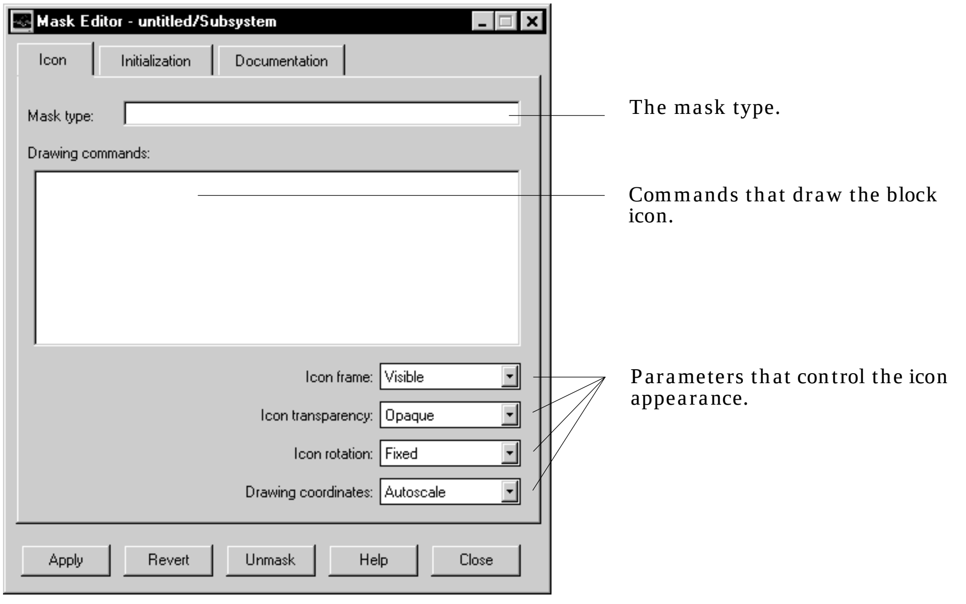
- Specify an initialization command without a terminating semicolon to echo its results to the command window.
- Place a `keyboard` command in the initialization commands to stop execution and give control to the keyboard. For more information, see the help text for the `keyboard` command.
- Enter either of these commands in the MATLAB command window:

  ```
  dbstop if error
  dbstop if warning
  ```

  If an error occurs in the initialization commands, execution stops and you can examine the mask workspace. For more information, see the help text for the `dbstop` command.

The Icon Page

The **Icon** page enables you to customize the masked block's icon. You create a custom icon by specifying commands in the **Drawing commands** field. You can create icons that show descriptive text, state equations, and graphics. This figure shows the **Icon** page:



Drawing commands have access to all variables in the mask workspace.

Drawing commands can display text, one or more plots, or show a transfer function. If you enter more than one command, the results of the commands are drawn on the icon in the order the commands appear.

Displaying Text on the Block Icon

To display text on the icon, enter one of these drawing commands:

```
disp('text') or disp(variablename)

text(x, y, 'text') or text(x, y, stringvariablename)

fprintf('text') or fprintf('format', variablename)
```

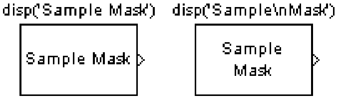
The `disp` command displays `text` or the contents of `variablename` centered on the icon.

The `text` command places a character string (`text` or the contents of `stringvariablename`) at a location specified by the point `(x,y)`. The units depend on the **Drawing coordinates** parameter. For more information, see “Controlling Icon Properties” on page 6–20.

The `fprintf` command displays formatted text centered on the icon and can display `text` along with the contents of `variablename`.

NOTE While these commands are identical in name to their corresponding MATLAB functions, they provide only the functionality described above.

To display more than one line of text, use `\n` to indicate a line break. For example, the figure below shows two samples of the `disp` command:



Displaying Graphics on the Block Icon

You can display plots on your masked block icon by entering one or more `plot` commands. You can use these forms of the plot command:

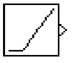
```
plot(Y);
plot(X1,Y1,X2,Y2,...);
```

`plot(Y)` plots, for a vector `Y`, each element against its index. If `Y` is a matrix, it plots each column of the matrix as though it were a vector.

`plot(X1,Y1,X2,Y2,...)` plots the vectors `Y1` against `X1`, `Y2` against `X2`, and so on. Vector pairs must be the same length and the list must consist of an even number of vectors.

For example, this command generates the plot that appears on the icon for the Ramp block, in the Sources library. The icon appears below the command:

```
plot([0 1 5], [0 0 4])
```



Plot commands can include `NaN` and `inf` values. When `NaN`s or `inf`s are encountered, Simulink stops drawing, then begins redrawing at the next numbers that are not `NaN` or `inf`.

The appearance of the plot on the icon depends on the value of the **Drawing coordinates** parameter. For more information, see "Controlling Icon Properties" on page 6–20.

Simulink displays three question marks (? ? ?) in the block icon and issues warnings in these situations:

- When the values for the parameters used in the drawing commands are not yet defined (for example, when the mask is first created and values have not yet been entered into the mask dialog box).
- When a masked block parameter or drawing command is entered incorrectly.

Displaying a Transfer Function on the Block Icon

To display a transfer function equation in the block icon, enter the following command in the **Drawing commands** field:

```
dpoly(num, den)
dpoly(num, den, 'character')
```

`num` and `den` are vectors of transfer function numerator and denominator coefficients, typically defined using initialization commands. The equation is expressed in terms of the specified `character`. The default is `s`. When the icon is drawn, the initialization commands are executed and the resulting equation is drawn on the icon.

- To display a continuous transfer function in descending powers of s, enter:

  ```
  dpoly(num, den)
  ```

For example, for `num = [0 0 1];` and `den = [1 2 1];` the icon looks like this:

$$\frac{1}{s^2+2s+1}$$

- To display a discrete transfer function in descending powers of z, enter:

  ```
  dpoly(num, den, 'z')
  ```

  For example, for `num = [0 0 1];` and `den = [1 2 1];` the icon looks like this:

$$\frac{1}{z^2+2z+1}$$

- To display a discrete transfer function in ascending powers of $1/z$, enter:

  ```
  dpoly(num, den, 'z-')
  ```

  For example, for `num` and `den` as defined above, the icon looks like this:

$$\frac{z^{-2}}{1+2z^{-1}+z^{-2}}$$

- To display a zero-pole gain transfer function, enter this command:

  ```
  droots(z, p, k)
  ```

  For example, the above command creates this icon for these values:

  ```
  z = []; p = [-1 -1]; k = 1;
  ```

$$\frac{1}{(s+1)(s+1)}$$

  You can add a fourth argument (`'z'` or `'z-'`) to express the equation in terms of z or $1/z$.

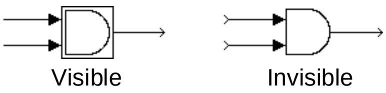
If the parameters are not defined or have no values when you create the icon, Simulink displays three question marks (? ? ?) in the icon. When the parameter values are entered in the mask dialog box, Simulink evaluates the transfer function and displays the resulting equation in the icon.

Controlling Icon Properties

You can control a masked block's icon properties by selecting among the choices below the **Drawing commands** field:

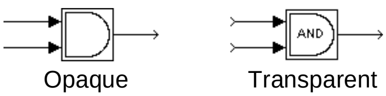
Icon frame

The icon frame is the rectangle that encloses the block. You can choose to show or hide the frame by setting the **Icon frame** parameter to **Visible** or **Invisible**. The default is to make the icon frame visible. For example, this figure shows visible and invisible icon frames for an AND gate block:



Icon transparency

The icon can be set to **Opaque** or **Transparent**, either hiding or showing what is underneath the icon. **Opaque**, the default, covers information Simulink draws, such as port labels. This figure shows opaque and transparent icons for an AND gate block. Notice the text on the transparent icon:

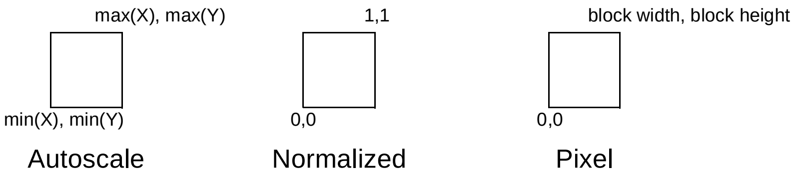


Icon rotation

When the block is rotated or flipped, you can choose whether to rotate or flip the icon, or to have it remain fixed in its original orientation. The default is not to rotate the icon. The icon rotation is consistent with block port rotation. This figure shows the results of choosing **Fixed** and **Rotates** icon rotation when the AND gate block is rotated:

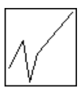
Drawing coordinates

This parameter controls the coordinate system used by the drawing commands. This parameter applies only to `plot` and `text` drawing commands. You can select from among these choices: **Autoscale**, **Normalized**, and **Pixel**:



- **Autoscale** automatically scales the icon within the block frame. When the block is resized, the icon is also resized. For example, this figure shows the icon drawn using these vectors:

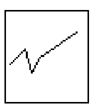
  ```
  X = [0 2 3 4 9]; Y = [4 6 3 5 8];
  ```

  The lower-left corner of the block frame is (0,3) and the upper-right corner is (9,8). The range of the *x*-axis is 9 (from 0 to 9), while the range of the *y*-axis is 5 (from 3 to 8).

- **Normalized** draws the icon within a block frame whose bottom-left corner is (0,0) and whose top right corner is (1,1). Only X and Y values between 0 and 1 appear. When the block is resized, the icon is also resized. For example, this figure shows the icon drawn using these vectors:

  ```
  X = [.0 .2 .3 .4 .9]; Y = [.4 .6 .3 .5 .8];
  ```

- **Pixel** draws the icon with X and Y values expressed in pixels. The icon is not automatically resized when the block is resized. To force the icon to resize with the block, define the drawing commands in terms of the block size.

  This example demonstrates how to create an improved icon for the `mx + b` sample masked subsystem discussed earlier in this chapter. These

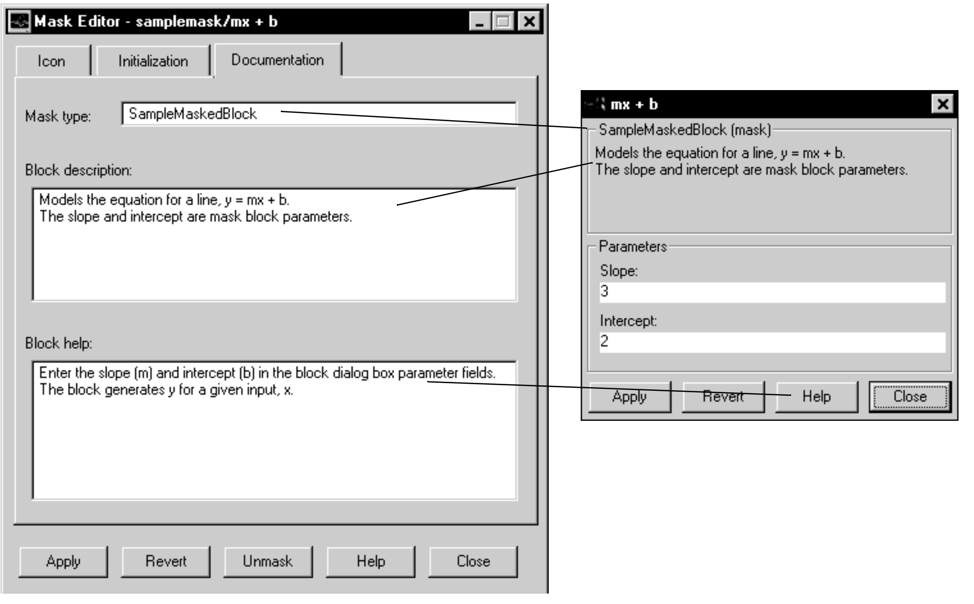
initialization commands define the data that enables the drawing command to produce an accurate icon regardless of the shape of the block.

```
pos = get_param(gcb, 'Position');
width = pos(3) – pos(1); height = pos(4) – pos(2);
x = [0, width];
if (m >= 0), y = [0, (m*width)]; end
if (m < 0),  y = [height, (height + (m*width))]; end
```

The drawing command that generates this icon is `plot(x,y)`.

The Documentation Page

The **Documentation** page enables you to define or modify the type, description, and help text for a masked block. This figure shows how fields on the **Documentation** page correspond to the mx+b sample mask block's dialog box.



The Mask Type Field

The mask type is a block classification used only for purposes of documentation. It appears in the block's dialog box and on all Mask Editor pages for the block. You can choose any name you want for the mask type. When Simulink creates the block's dialog box, it adds "(mask)" after the mask type to differentiate masked blocks from built-in blocks.

The Block Description Field

The block description is informative text that appears in the block's dialog box in the frame under the mask type. If you are designing a system for others to use, this is a good place to describe the block's purpose or function.

Simulink automatically wraps long lines of text. You can force line breaks by using the **Enter** or **Return** key.

The Mask Help Text Field

You can provide help text that gets displayed when the **Help** button is pressed on the masked block's dialog box. If you create models for others to use, this is a good place to explain how the block works and how to enter its parameters.

You can include user-written documentation for a masked block's help. You can specify any of the following for the masked block help text:

- URL specification (a string starting with `http:`, `www`, `file:`, `ftp:`, or `mailto:`)
- `web` command (launches a browser)
- `eval` command (evaluates a MATLAB string)
- Static text displayed in the web browser

Simulink examines the first line of the masked block help text. If it detects a URL specification, `web` command, or `eval` command, it accesses the block help as directed; otherwise, the full contents of the masked block help text are displayed in the browser.

These examples illustrate several acceptable commands:

```
web([docroot '/My Blockset Doc/' get_param(gcb,'MaskType') '.html'])
eval('!Word My_Spec.doc')
http://www.mathworks.com
file:///c:/mydir/helpdoc.html
www.mathworks.com
```

Simulink automatically wraps long lines of text.

7

Conditionally Executed Subsystems

Introduction

A *conditionally executed subsystem* is a subsystem whose execution depends on the value of an input signal. The signal that controls whether a subsystem executes is called the *control signal*. The signal enters the Subsystem block at the *control input*.

Conditionally executed subsystems can be very useful when building complex models that contain components whose execution depends on other components.

Simulink supports three types of conditionally executed subsystems:

- An *enabled subsystem* executes while the control signal is positive. It starts execution at the time step where the control signal crosses zero (from the negative to the positive direction) and continues execution while the control signal remains positive. Enabled subsystems are described in more detail on page 7–3.
- A *triggered subsystem* executes once each time a “trigger event” occurs. A trigger event can occur on the rising or falling edge of a trigger signal, which can be continuous or discrete. Triggered subsystems are described in more detail on page 7–8.
- A *triggered and enabled subsystem* executes once on the time step when a trigger event occurs if the enable control signal has a positive value at that step. Triggered and enabled subsystems are described in more detail on page 7–11.

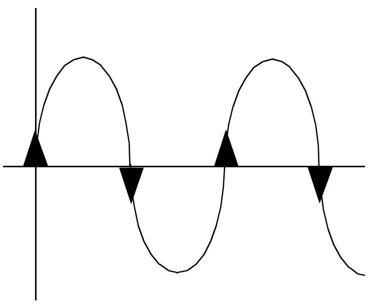
Enabled Subsystems

Enabled subsystems are subsystems that execute at each simulation step where the control signal has a positive value.

An enabled subsystem has a single control input, which can be scalar or vector valued:

- If the input is a scalar, the subsystem executes if the input value is greater than zero.
- If the input is a vector, the subsystem executes if *any* of the vector elements is greater than zero.

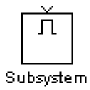
For example, if the control input signal is a sine wave, the subsystem is alternately enabled and disabled, as shown in this figure. An up arrow signifies enable, a down arrow disable:



Simulink uses the zero-crossing slope method to determine whether an enable is to occur. If the signal crosses zero and the slope is positive, the subsystem is enabled. If the slope is negative at the zero crossing, the subsystem is disabled.

Creating an Enabled Subsystem

You create an enabled subsystem by copying an Enable block from the Connections library into a subsystem. Simulink adds an enable symbol and an enable control input port to the Subsystem block icon:

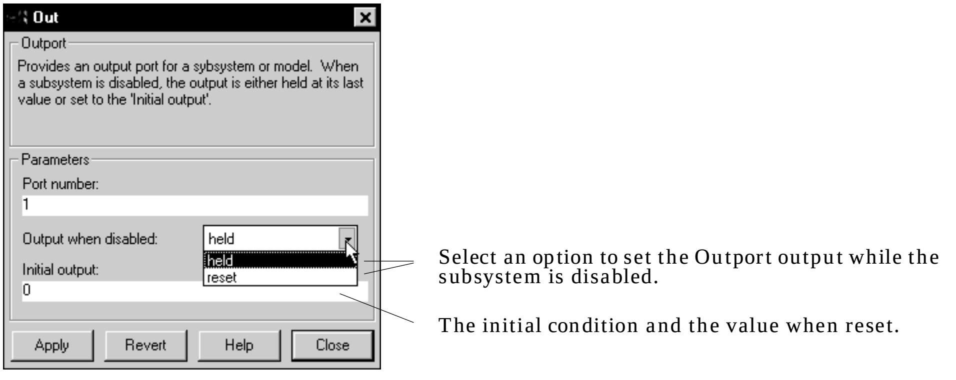


Setting Output Values while the Subsystem Is Disabled

Although an enabled subsystem does not execute while it is disabled, the output signal is still available to other blocks. While an enabled subsystem is disabled, you can choose to hold the subsystem outputs at their previous values or reset them to their initial conditions.

Open each Outport block's dialog box and select one of the choices for the **Output when disabled** parameter, as shown in the dialog box below:

- Choose **held** to cause the output to maintain its most recent value.
- Choose **reset** to cause the output to revert to its initial condition. Set the **Initial output** to the initial value of the output.

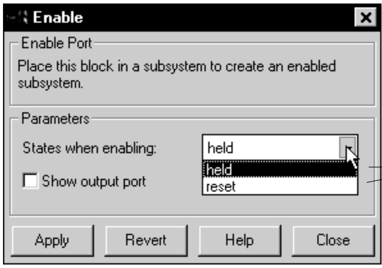


Setting States When the Subsystem Becomes Re-enabled

When an enabled subsystem executes, you can choose whether to hold the subsystem states at their previous values or reset them to their initial conditions.

To do this, open the Enable block dialog box and select one of the choices for the **States when enabling** parameter, as shown in the dialog box below:

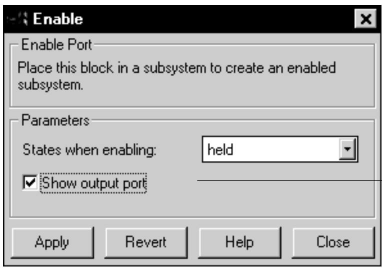
- Choose **held** to cause the states to maintain their most recent values.
- Choose **reset** to cause the states to revert to their initial conditions.



Select an option to set the states when the subsystem is re-enabled.

Outputting the Enable Control Signal

An option on the Enable block dialog box lets you output the enable control signal. To output the control signal, select the **Show output port** check box:



Select this check box to show the output port.

This feature allows you to pass the control signal down into the enabled subsystem, which can be useful where logic within the enabled subsystem is dependent on the value or values contained in the control signal.

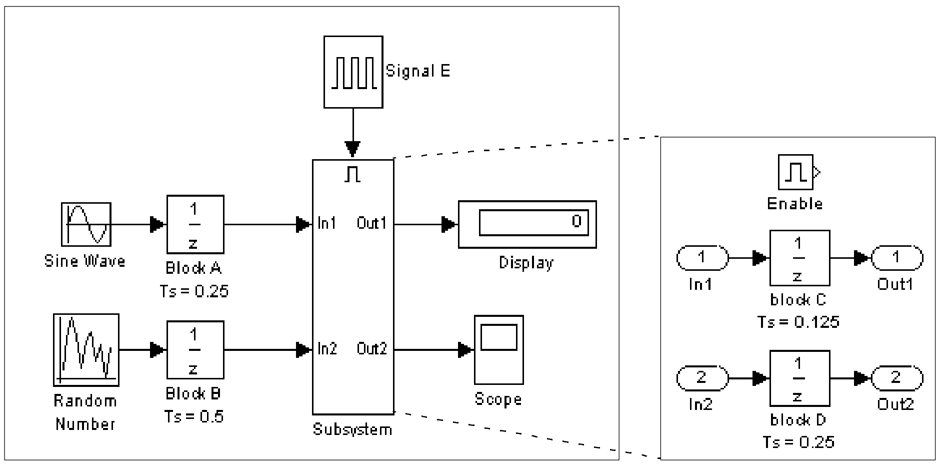
Blocks an Enabled Subsystem Can Contain

An enabled subsystem can contain any block, whether continuous or discrete. Discrete blocks in an enabled subsystem execute only when the subsystem executes, and only when their sample times are synchronized with the simulation sample time. Enabled subsystems and the model use a common clock.

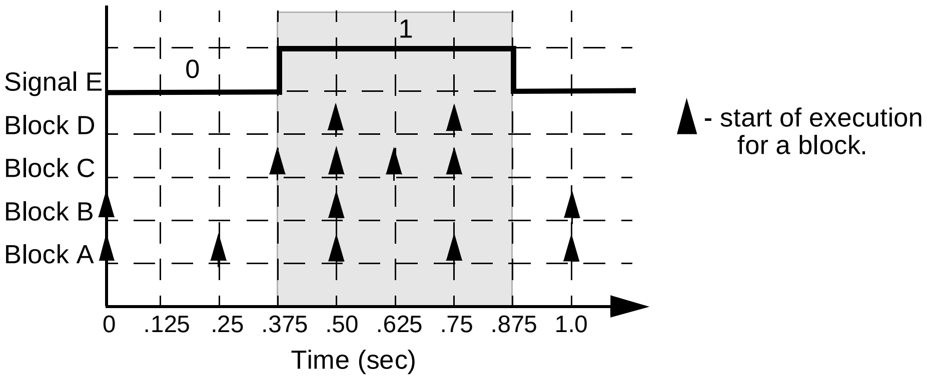
For example, this system contains four discrete blocks and a control signal. The discrete blocks are:

- Block A, which has a sample time of 0.25 seconds.
- Block B, which has a sample time of 0.5 seconds.
- Block C, within the Enabled subsystem, which has a sample time of 0.125 seconds.
- Block D, also within the Enabled subsystem, which has a sample time of 0.25 seconds.

The enable control signal is generated by a Pulse Generator block, labeled Signal E, which changes from 0 to 1 at 0.375 seconds and returns to 0 at 0.875 seconds.



The chart below indicates when the discrete blocks execute:



Blocks A and B execute independent of the enable signal because they are not part of the enabled subsystem. When the enable signal becomes positive, blocks C and D execute at their assigned sample rates until the enable signal becomes zero again. Note that block C does not execute at 0.875 seconds when the enable signal changes to zero.

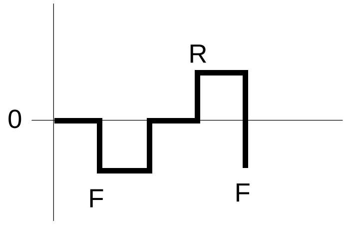
Triggered Subsystems

Triggered subsystems are subsystems that execute each time a trigger event occurs.

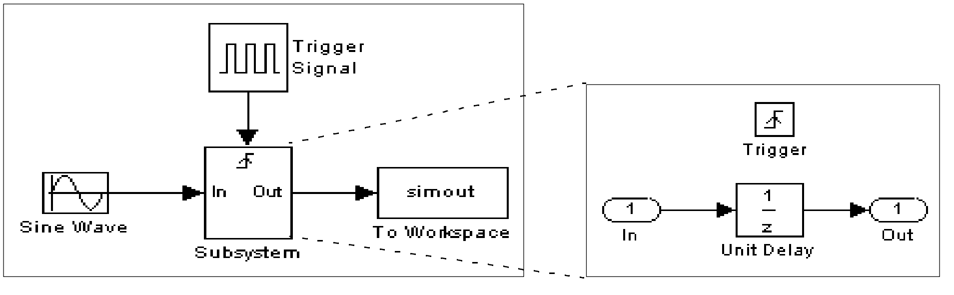
A triggered subsystem has a single control input, called the *trigger input*, which determines whether the subsystem executes. You can choose from three types of trigger events to force a triggered subsystem to begin execution:

- A *rising trigger*, where the control signal, or any element of a vector control signal, changes from 0 to a positive value.
- A *falling trigger*, where the control signal, or any element of a vector control signal, changes from 0 to a negative value.
- Either a rising or a falling trigger, where the control signal, or any element of a vector control signal, changes from 0 to a nonzero value.

For example, this figure shows when rising (R) and falling (F) triggers occur for the given control signal:



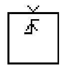
A simple example of a trigger subsystem is illustrated below:



In this example, the subsystem is triggered on the rising edge of the square wave trigger control signal.

Creating a Triggered Subsystem

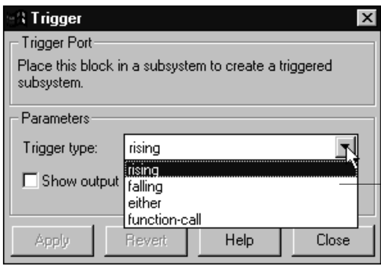
You create a triggered subsystem by copying the Trigger block from the Connections library into a subsystem. Simulink adds a trigger symbol and a trigger control input port to the Subsystem block icon:



Subsystem

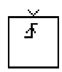
To select the trigger type, open the Trigger block dialog box and select one of the choices for the **Trigger type** parameter, as shown in the dialog box below:

- **rising** forces a trigger whenever the trigger signal crosses zero in a positive direction.
- **falling** forces a trigger whenever the trigger signal crosses zero in a negative direction.
- **either** forces a trigger whenever the trigger signal crosses zero in either direction.

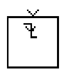


Select the trigger type from these choices.

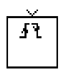
Simulink uses different symbols on the Trigger and Subsystem blocks to indicate rising and falling triggers (or either). This figure shows the trigger symbols on Subsystem blocks:



Subsystem with Rising trigger



Subsystem with Falling trigger



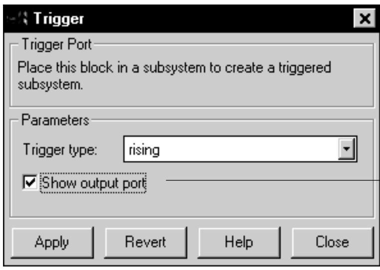
Subsystem with Rising or Falling trigger

Outputs and States between Trigger Events

Unlike enabled subsystems, triggered subsystems always hold their outputs at the last value between triggering events. Also, triggered subsystems cannot reset their states when triggered; states of any discrete blocks are held between trigger events.

Outputting the Trigger Control Signal

An option on the Trigger block dialog box lets you output the trigger control signal. To output the control signal, select the **Show output port** check box:



Select this check box to show the output port.

Function-Call Subsystems

You can create a triggered subsystem whose execution is determined by logic internal to an S-function instead of by the value of a signal. These subsystems are called *function-call subsystems*. For more information about function-call subsystems, see "Using Function-Call Subsystems" on page 8–51.

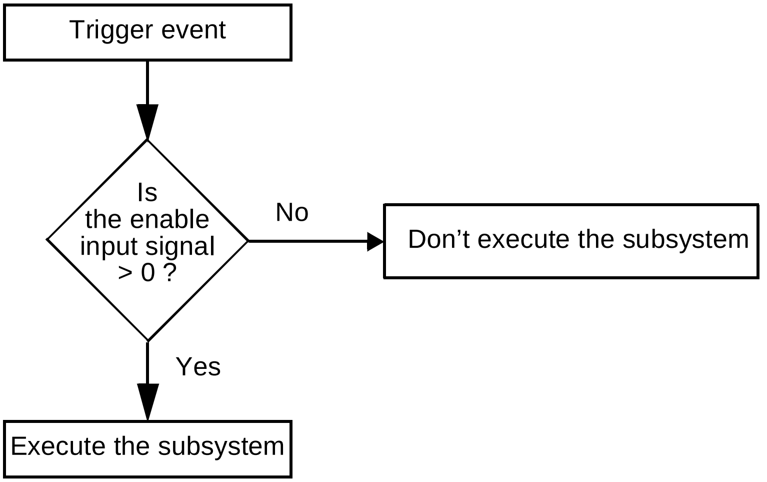
Blocks a Triggered Subsystem Can Contain

Triggered systems only execute at specific instances during a simulation. As a result, the only blocks that are suitable for use in a triggered subsystem are:

- Blocks with inherited sample time, such as the Logical Operator block or the Gain block.
- Discrete blocks having their sample time set to –1, which indicates that the sample time is inherited from the driving block.

Triggered and Enabled Subsystems

A third kind of conditionally executed subsystem combines both types of conditional execution. The behavior of this type of subsystem, called a *triggered and enabled* subsystem, is a combination of the enabled subsystem and the triggered subsystem, as shown by this flow diagram:

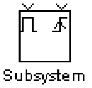


A triggered and enabled subsystem contains both an enable input port and a trigger input port. When the trigger event occurs, Simulink checks the enable input port to evaluate the enable control signal. If its value is greater than zero, Simulink executes the subsystem. If both inputs are vectors, the subsystem executes if at least one element of each vector is nonzero.

The subsystem executes once at the time step at which the trigger event occurs.

Creating a Triggered and Enabled Subsystem

You create a triggered and enabled subsystem by dragging both the Enable and Trigger blocks from the Connections library into an existing subsystem. Simulink adds enable and trigger symbols and enable and trigger and enable control inputs to the Subsystem block icon:

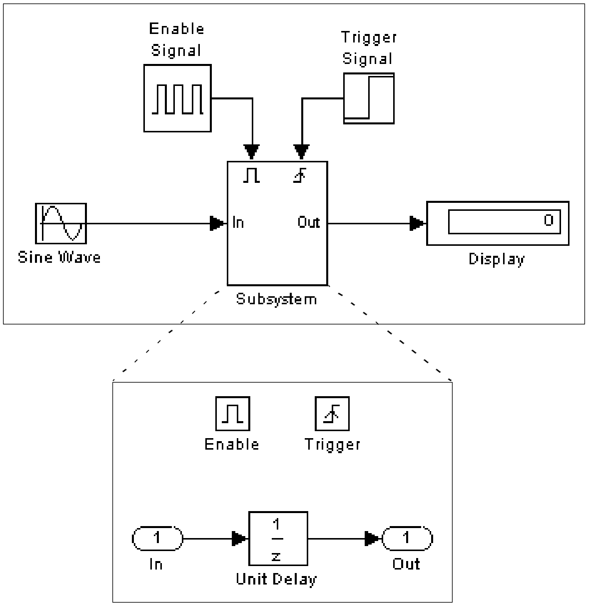


You can set output values when a triggered and enabled subsystem is disabled as you would for an enabled subsystem. For more information, see “Setting Output Values while the Subsystem Is Disabled” on page 7–4. Also, you can specify what the values of the states are when the subsystem is re-enabled. See “Setting States When the Subsystem Becomes Re-enabled” on page 7–4.

Set the parameters for the Enable and Trigger blocks separately. The procedures are the same as those described for the individual blocks.

A Sample Triggered and Enabled Subsystem

A simple example of a triggered and enabled subsystem is illustrated in the model below.



8

S-Functions

Introduction

S-functions (System-functions) provide a powerful mechanism for augmenting and extending Simulink's capabilities. The introductory sections of this chapter describe what an S-function is and when and why you might use one. This chapter then presents a comprehensive description of how to write your own S-functions.

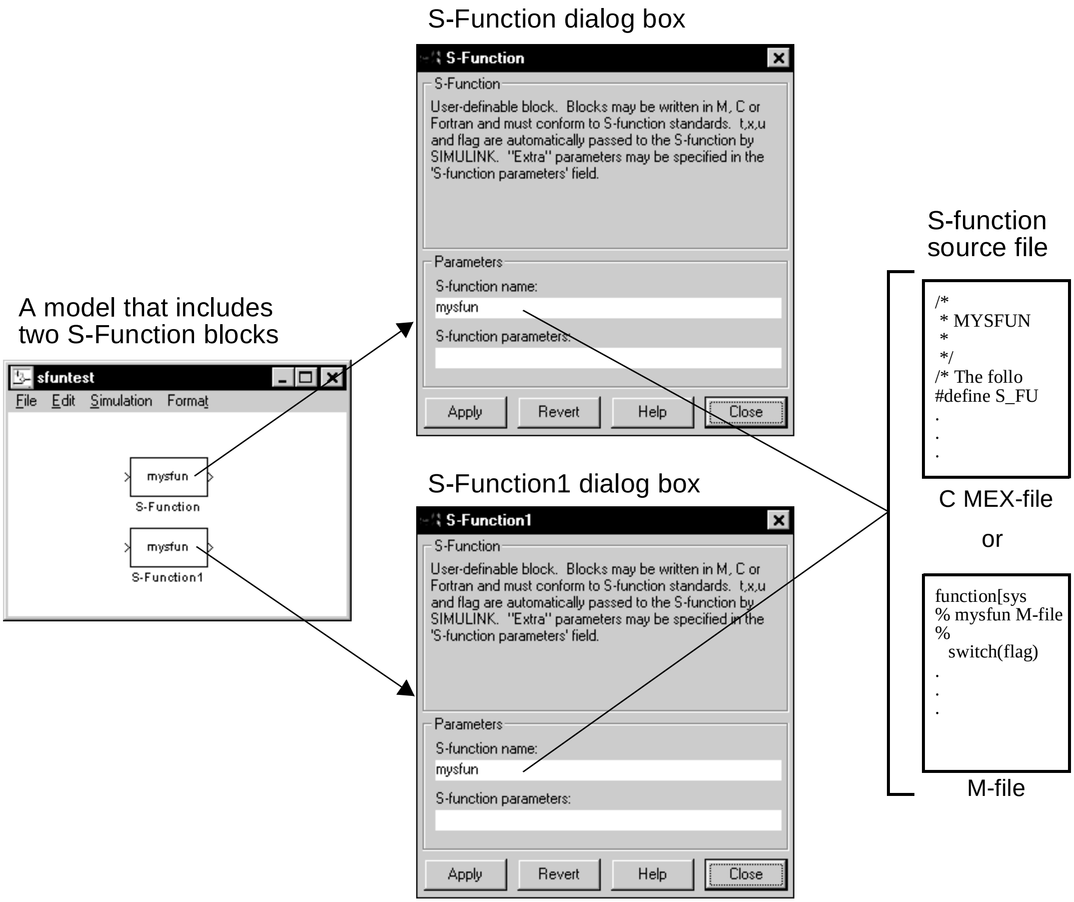
What Is an S-Function?

An *S-function* is a programmatic description of a dynamic system. S-functions can be written using MATLAB or C. C language S-functions are compiled as MEX-files using the `mex` utility described in the *Application Program Interface Guide*. As with other MEX-files, they are dynamically linked into MATLAB when needed.

S-functions use a special calling syntax that enables you to interact with the ODE solvers. This interaction is very similar to the interaction that takes place between the solvers and built-in Simulink blocks.

The form of an S-function is very general and can accommodate continuous, discrete, and hybrid systems. As a result, nearly all Simulink models can be described as S-functions.

S-functions are incorporated into Simulink models by using the S-Function block in the Nonlinear Library. Use the S-Function block's dialog box to specify the name of the underlying S-function, as illustrated in the figure below:



This illustration shows the relationship between an S-Function block, its dialog box, and the source file that defines the block's behavior. In this example, the model contains two instances of an S-Function block. Both blocks reference the same source file (`mysfun`, which can be either a C MEX-file or an M-file). If both a C MEX-file and an M-file exist with the same name, the C MEX-file takes precedence and is the file that the S-function uses.

You can use Simulink's masking facility to create custom dialog boxes and icons for your S-Function blocks. Masked dialog boxes can make it easier to specify *additional parameters* for S-functions. For a discussion of additional parameters, see "Passing Additional Parameters" on page 8–25 and

"Specifying Parameter Values Interactively" on page 8–43. For more information about masking, see Chapter 6.

When To Use an S-Function

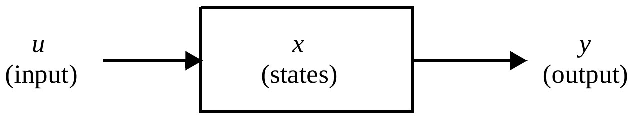
The most common use of S-functions is to create custom Simulink blocks. You can use S-functions for a variety of applications, including:

- Adding new general purpose blocks to Simulink
- Incorporating existing C code into a simulation
- Describing a system as a mathematical set of equations
- Using graphical animations (see the inverted pendulum demo, `penddemo`)

An advantage of using S-functions is that you can build a general purpose block that you can use many times in a model, varying parameters with each instance of the block.

How S-Functions Work

Each block within a Simulink model has the following general characteristics: a vector of inputs, u, a vector of outputs, y, and a vector of states, x, as shown by this illustration:



The state vector may consist of continuous states, discrete states, or a combination of both. The mathematical relationships between the inputs, outputs, and the states are expressed by the following equations:

$$y = f_o(t,x,u) \qquad \texttt{(output)}$$

$$\dot{x}_c = f_d(t,x,u) \qquad \texttt{(derivative)}$$

$$x_{d_{k+1}} = f_u(t,x,u) \qquad \texttt{(update)}$$

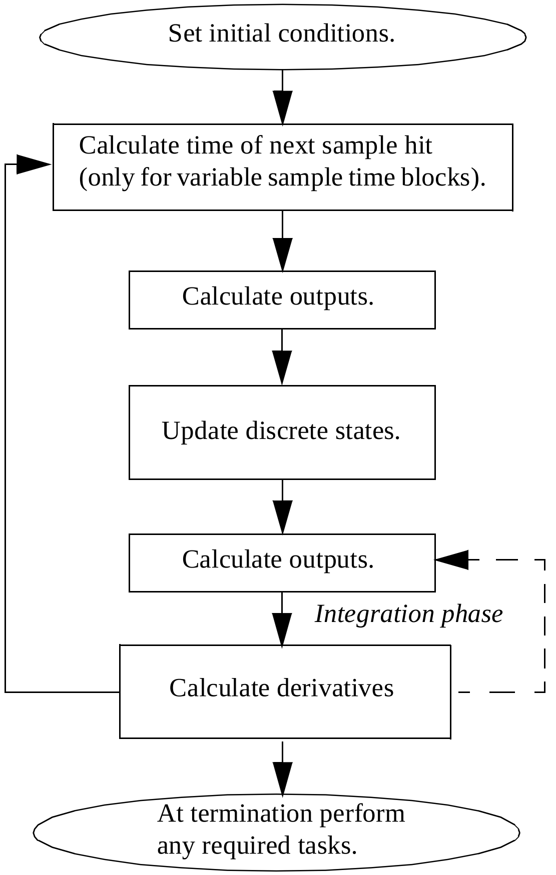
where $x = \begin{bmatrix} x_c \\ x_{d_k} \end{bmatrix}$

Simulink partitions the state vector into two parts: the continuous states and the discrete states. The continuous states occupy the first part of the state vector, and the discrete states occupy the second part. For blocks with no states, x is an empty vector.

Simulation Stages and S-Function Routines

Simulink makes repeated calls during specific stages of simulation to each block in the model, directing it to perform tasks such as computing its outputs, updating its discrete states, or computing its derivatives. Additional calls are made at the beginning and end of a simulation to perform initialization and termination tasks.

This figure shows the order in which Simulink performs the simulation stages:



Simulink makes repeated calls to *S-function routines*, which perform the tasks required at each stage. All the S-function routines begin with the prefix `mdl`.

In M-file S-functions, the S-function routines are implemented as M-file subfunctions. In C MEX-file S-functions, they are implemented as C subroutines. The names of the S-function routines and the functions they perform are identical in M-file and C MEX-file S-functions.

For an M-file S-function, Simulink passes a `flag` parameter to the S-function. The `flag` indicates the current simulation stage. You must write M-code that calls the appropriate functions for each `flag` value. For a C MEX-file

S-function, Simulink calls the S-function routines directly. This table lists the simulation stages, the corresponding S-function routines, and the associated `flag` value for M-file S-functions:

Table 8-1: Simulation Stages

Simulation Stage	S-Function Routine	Flag (M-File S-Functions)
Initialization	`mdlInitializeSizes`	`flag = 0`
Calculation of next sample hit (optional)	`mdlGetTimeOfNextVarHit`	`flag = 4`
Calculation of outputs	`mdlOutputs`	`flag = 3`
Update discrete states	`mdlUpdate`	`flag = 2`
Calculation of derivatives	`mdlDerivatives`	`flag = 1`
End of simulation tasks	`mdlTerminate`	`flag = 9`

C MEX-file S-function routines must have exactly the names shown in the figure above. In M-file S-functions, you must provide code that, based on the `flag` value, calls the appropriate S-function routine. A template M-file S-function, `sfuntmpl.m`, is located in `/toolbox/simulink/blocks`. This template uses a `switch` statement to handle the `flag` values. All you have to do is place your code in the correct S-function routine.

In C MEX-file S-functions, Simulink directly calls the correct S-function routine for the current simulation stage. A template S-function written in C called `sfuntmpl.c`, located under `simulink/src`, is supplied with Simulink. For a more amply commented version of the template, see `sfuntmpl.doc` in the same directory.

NOTE We recommend that you use the M-file or C MEX-file template when developing S-functions.

S-Function Concepts

Understanding these key concepts should enable you to build S-functions correctly:

- Direct feedthrough
- Dynamically sized inputs
- Setting sample times and offsets
- Initial sample times

Direct Feedthrough

Direct feedthrough means that the output or the variable sample time is controlled directly by the value of an input port. A good rule of thumb is that an S-function has direct feedthrough if:

- Its output (`mdlOutputs`) is a function of the input `u`. Outputs may also include graphical outputs, as in the case of an XY Graph scope.
- It is a variable sample time S-function (calls `mdlGetTimeOfNextVarHit`) and the computation of the next sample hit requires the input `u`.

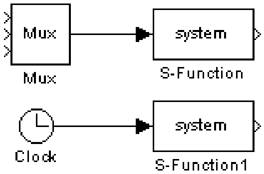
It is very important to set the direct feedthrough flag correctly because it affects the execution order of the blocks in your model and is used to detect algebraic loops.

Dynamically Sized Inputs

S-functions can be written to support arbitrary width inputs. In this case, the actual input width is determined dynamically when a simulation is started by evaluating the width of the input vector driving the S-function. The input width can also be used to determine the number of continuous states, the number of discrete states, and the number of outputs.

To indicate that the input width is dynamically sized, specify a value of -1 for the appropriate fields in the `sizes` structure, which is returned during the `mdlInitialize` call. You can determine the actual input width when your S-function is called by using `length(u)`.

For example, the illustration below shows two instances of the same S-Function block in a model.



The upper S-Function block is driven by a block with a three-element output vector. The lower S-Function block is driven by a block with a scalar output. By specifying that the S-Function block has dynamically sized inputs, the same S-function can accommodate both situations. Simulink automatically calls the block with the appropriately sized input vector. Similarly, if other block characteristics, such as the number of outputs or the number of discrete or continuous states, are specified as dynamically sized, Simulink defines these vectors to be the same length as the input vector.

Setting Sample Times and Offsets

If the S-Function block's behavior is a function of discrete time intervals, you can define a sample time to control when Simulink calls the block. You can also define an offset that delays each sample time hit. The value of the offset cannot exceed the corresponding sample time.

A *sample time hit* occurs at time values determined by this formula:

```
time = (n * sample time value) + offset
```

where `n` is an integer whose first value is zero.

When a sample time is specified, Simulink calls the `mdlOutput` and `mdlUpdate` functions at each sample time hit, as defined by the above equation.

When writing a single rate discrete S-function, it is not necessary to put explicit tests for sample time hits in your output and update functions. Simulink calls your S-function only at the appropriate sample hits. Refer to either the M-file or C MEX-file hybrid example (page 8-20 and page 8-20, respectively) for how to handle multiple sample times.

Inherited Sample Times

Sometimes an S-Function block has no inherent sample time characteristics (that is, it is either continuous or discrete, depending on the sample time of

some other block in the system). You can specify that the block's sample time is *inherited*. A simple example of this is a Gain block that inherits its sample time from the block driving it.

A block can inherit its sample time from:

- The driving block
- The destination block
- The fastest sample time in the system

To set a block's sample time as inherited, use `-1` as the sample time. For more information on the propagation of sample times, see "Sample Time Colors" on page 10–13.

Sample S-Functions

It is helpful to examine some sample S-functions as you read the next sections. Other examples are stored in these subdirectories under the MATLAB root directory:

- C MEX-files: `simulink/src`
- M-files: `toolbox/simulink/blocks`

The `simulink/blocks` directory contains many M-file S-functions. Consider starting off by looking at these files: `csfunc.m`, `dsfunc.m`, `vsfunc.m`, `mixed.m`, `vdpm.m`, `simom.m`, `simom2.m`, `limintm.m`, `vlimintm.m`, and `vdlmintm.m`.

Writing S-Functions as M-Files

An M-file that defines an S-Function block must provide information about the model; Simulink needs that information during simulation. As the simulation proceeds, Simulink, the ODE solver, and the M-file interact to perform specific tasks. These tasks include defining initial conditions and block characteristics, and computing derivatives, discrete states, and outputs.

Simulink provides a template M-file S-function that includes statements that define necessary functions, as well as comments to help you write the code needed for your S-function block. This template file, `sfuntmpl.m`, is in the directory `toolbox/simulink/blocks` below the MATLAB root directory.

Defining S-Function Block Characteristics

For Simulink to recognize an M-file S-function, you must provide it with specific information about the S-function. This information includes the number of inputs, outputs, states, and other block characteristics.

To give Simulink this information, call the `simsizes` function at the beginning of `mdlInitializeSizes`:

```
sizes = simsizes;
```

This function returns an unpopulated `sizes` structure. The table below lists the `sizes` structure fields and describes the information contained in each field:

Table 8-2: Fields in the sizes Structure

Field Name	Description
`sizes.NumContStates`	Number of continuous states
`sizes.NumDiscStates`	Number of discrete states
`sizes.NumOutputs`	Number of outputs
`sizes.NumInputs`	Number of inputs
`sizes.DirFeedthrough`	Flag for direct feedthrough
`sizes.NumSampleTimes`	Number of sample times

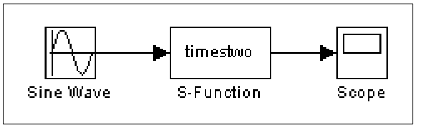
After you initialize the sizes structure, call simsizes again:

```
sys = simsizes(sizes);
```

This passes the information in the sizes structure to sys, a vector that holds the information for use by Simulink.

A Simple M-File S-Function Example

The easiest way to understand how S-functions work is to look at a simple example. This block takes an input scalar signal and doubles it:



The M-file code that contains the S-function is modeled on an S-function template called sfuntmpl.m, which is included with Simulink. By using the template, you can create an M-file S-function that is very close in appearance to a C MEX-file S-function. This is useful because it makes a transition from an M-file to an C MEX-file much easier.

Below is the M-file code for the timestwo.m block:

```
function [sys,x0,str,ts] = timestwo(t,x,u,flag)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

   case 0
     [sys,x0,str,ts] = mdlInitializeSizes; % Initialization

   case 3
     sys = mdlOutputs(t,x,u); % Calculate outputs

   case { 1, 2, 4, 9 }
     sys = []; % Unused flags

   otherwise
     error(['Unhandled flag = ',num2str(flag)]); % Error handling
% End of function timestwo
```

The first four input arguments, which Simulink passes to the S-function, must be the variables `t`, `x`, `u`, and `flag`:

- `t`, the time
- `x`, the state vector (required even if, as in this case, there are no states)
- `u`, the input vector
- `flag`, the parameter that controls the S-function subroutine calls at each simulation stage

Simulink also requires that the output parameters, `sys`, `x0`, `str`, and `ts` be placed in the order given. These parameters are:

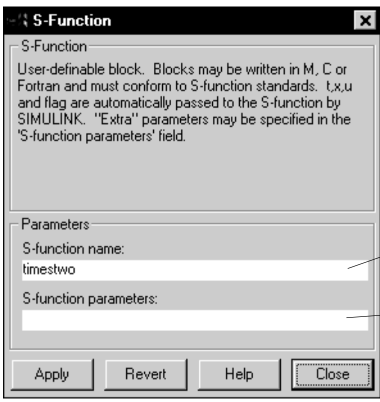
- `sys`, a generic return argument. The values returned depend on the flag value. For example, for `flag = 3`, `sys` contains the S-function outputs.
- `x0`, the initial state values (an empty vector if there are no states in the system).
- `str`, provided only for consistency with the S-function API for block diagrams. For M-file S-functions, set it to an empty matrix.
- `ts`, a two column matrix containing the sample times and offsets of states associated with the block. Sample times should be declared in ascending order. Continuous systems have their sample time set to zero. The hybrid example, which starts on page 8-20, demonstrates an S-function with multiple sample times.

Below are the S-function subroutines that `timestwo.m` calls:

```
%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    1;
sizes.NumInputs=     1;
sizes.DirFeedthrough=0;
```

```
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [-1 0]; % Inherited sample time
% end of mdlInitializeSizes
%==============================================================
% Function mdlOutputs performs the calculations.
%==============================================================
function sys = mdlOutputs(t,x,u)
sys = 2*u;

% End of mdlOutputs.
```

To test this S-function in Simulink, connect a sine wave generator to the input of an S-Function block. Connect the output of the S-Function block to a Scope. Double-click on the S-Function block to open the dialog box:



Enter the function name here. In this example, type `timestwo`.

If you have additional parameters to pass to the block, enter their names here, separating them with commas. In this example, there are no additional parameters.

You can now run this simulation.

Examples of M-File S-Functions

The simple example discussed above has no states. Most S-Function blocks require the handling of states, whether continuous or discrete. The sections that follow discuss four common types of systems you can model in Simulink using S-functions:

- Continuous
- Discrete
- Hybrid
- Variable-step

All examples are based on the M-file S-function template found in `sfuntmpl.m`.

Example - Continuous State S-Function

Simulink includes a function called `csfunc.m`, which is an example of a continuous state system modeled in an S-function. Here is the code for the M-file S-function:

```
function [sys,x0,str,ts] = csfunc(t,x,u,flag)
% CSFUNC An example M-file S-function for defining a system of
% continuous state equations:
%      x' = Ax + Bu
%      y  = Cx + Du
%
% Generate a continuous linear system:
A=[-0.09   -0.01
    1          0];
B=[ 1   -7
    0   -2];
C=[ 0    2
    1   -5];
D=[-3    0
    1    0];
%
% Dispatch the flag.
%
switch flag,

  case 0
```

```
    [sys,x0,str,ts]=mdlInitializeSizes(A,B,C,D); % Initialization

  case 1
    sys = mdlDerivatives(t,x,u,A,B,C,D); % Calculate derivatives

  case 3
    sys = mdlOutputs(t,x,u,A,B,C,D); % Calculate outputs

  case { 2, 4, 9 } % Unused flags
    sys = [];

  otherwise
    error(['Unhandled flag = ',num2str(flag)]); % Error handling
end
% end csfunc
%
%==============================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the
% S-function.
%==============================================================
%
function [sys,x0,str,ts] = mdlInitializeSizes(A,B,C,D)
%
% call simsizes for a sizes structure, fill it in and convert it
% to a sizes array.
%
sizes = simsizes;
sizes.NumContStates  = 2;
sizes.NumDiscStates  = 0;
sizes.NumOutputs     = 2;
sizes.NumInputs      = 2;
sizes.DirFeedthrough = 1;     % Matrix D is nonempty.
sizes.NumSampleTimes = 1;
sys = simsizes(sizes);
%
% initialize the initial conditions
%
x0 = zeros(2,1);
%
```

```
% str is an empty matrix
%
str = [];
%
% Initialize the array of sample times; in this example the sample
% time is continuous, so set ts to 0 and its offset to 0.
%
ts = [0 0];
% end mdlInitializeSizes
%
%===============================================================
% mdlDerivatives
% Return the derivatives for the continuous states.
%===============================================================
function sys = mdlDerivatives(t,x,u,A,B,C,D)
sys = A*x + B*u;
% end mdlDerivatives
%
%===============================================================
% mdlOutputs
% Return the block outputs.
%===============================================================
%
function sys = mdlOutputs(t,x,u,A,B,C,D)
sys = C*x + D*u;
% end mdlOutputs
```

The above example conforms to the simulation stages discussed earlier in this chapter. Unlike `timestwo.m`, this example invokes `mdlDerivatives` to calculate the derivatives of the continuous state variables when `flag = 1`. The system state equations are of the form

```
x'= Ax + Bu
y = Cx + Du
```

so that very general sets of continuous differential equations can be modeled using `csfunc.m`. Note that `csfunc.m` is similar to the built-in State-Space block. This S-function can be used as a starting point for a block that models a state-space system with time-varying coefficients.

Example - Discrete State S-Function

Simulink includes a function called `dsfunc.m`, which is an example of a discrete state system modeled in an S-function. This function is similar to `csfunc.m`, the continuous state S-function example. The only difference is that `mdlUpdate` is called instead of `mdlDerivative`. `mdlUpdate` updates the discrete states when the `flag = 2`. Note that for a single-rate discrete S-function, Simulink calls the `mdlUpdate`, `mdlOutput`, and `mdlGetTimeOfNextVarHit` (if needed) routines only on sample hits. Here is the code for the M-file S-function:

```
function [sys,x0,str,ts] = dsfunc(t,x,u,flag)
% An example M-file S-function for defining a discrete system.
% This S-function implements discrete equations in this form:
%      x(n+1) = Ax(n) + Bu(n)
%      y(n)   = Cx(n) + Du(n)
%
% Generate a discrete linear system:
A=[-1.3839   -0.5097
    1.0000         0];
B=[-2.5559         0
         0    4.2382];
C=[      0    2.0761
         0    7.7891];
D=[   -0.8141   -2.9334
       1.2426         0];

switch flag
  case 0
    sys = mdlInitializeSizes(A,B,C,D); % Initialization

  case 2
    sys = mdlUpdate(t,x,u,A,B,C,D); % Update discrete states

  case 3
    sys = mdlOutputs(t,x,u,A,B,C,D); % Calculate outputs

  case {1, 4, 9} % Unused flags
    sys = [];

  otherwise
    error(['unhandled flag = ',num2str(flag)]); % Error handling
```

```
end
% end of dsfunc

%==============================================================
% Initialization
%==============================================================

function [sys,x0,str,ts] = mdlInitializeSizes(A,B,C,D)

% call simsizes for a sizes structure, fill it in, and convert it
% to a sizes array.

sizes = simsizes;
sizes.NumContStates  = 0;
sizes.NumDiscStates  = 2;
sizes.NumOutputs     = 2;
sizes.NumInputs      = 2;
sizes.DirFeedthrough = 1; % Matrix D is non-empty.
sizes.NumSampleTimes = 1;
sys = simsizes(sizes);
x0  = ones(2,1);   % Initialize the discrete states.
str = [];          % Set str to an empty matrix.
ts  = [1 0];       % sample time: [period, offset]
% end of mdlInitializeSizes

%==============================================================
% Update the discrete states
%==============================================================
function sys = mdlUpdates(t,x,u,A,B,C,D)
sys = A*x + B*u;
% end of mdlUpdate

%==============================================================
% Calculate outputs
%==============================================================
function sys = mdlOutputs(t,x,u,A,B,C,D)
sys = C*x + D*u;
% end of mdlOutputs
```

The above example conforms to the simulation stages discussed earlier in this chapter. The system discrete state equations are of the form

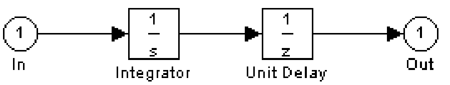
```
x(n+1) = Ax(n) + Bu(n)
y(n)   = Cx(n) + Du(n)
```

so that very general sets of difference equations can be modeled using `dsfunc.m`. This is similar to the built-in Discrete State-Space block. You can use `dsfunc.m` as a starting point for modeling discrete state-space systems with time-varying coefficients.

Example - Hybrid System S-Functions

Simulink includes a function called `mixedm.m`, which is an example of a hybrid system (a combination of continuous and discrete states) modeled in an S-function. Handling hybrid systems is fairly straightforward; the `flag` parameter forces the calls to the correct S-function subroutine for the continuous and discrete parts of the system. One subtlety of hybrid S-functions (or any multirate S-function) is that Simulink calls the `mdlUpdate`, `mdlOutput`, and `mdlGetTimeOfNextVarHit` routines at all sample times. This means that in these routines you must test to determine which sample hit is being processed and only perform updates that correspond to that sample hit.

`mixed.m` models a continuous Integrator followed by a discrete Unit Delay. In Simulink block diagram form, the model looks like this:



Here is the code for the M-file S-function:

```
function [sys,x0,str,ts] = mixedm(t,x,u,flag)
% A hybrid system example that implements a hybrid system
% consisting of a continuous integrator (1/s) in series with a
% unit delay (1/z).
%
% Set the sampling period and offset for unit delay.
dperiod = 1;
doffset = 0;
switch flag

  case 0         % Initialization
```

```
    [sys,x0,str,ts] = mdlInitializeSizes(dperiod,doffset);

  case 1
    sys = mdlDerivatives(t,x,u); % Calculate derivatives

  case 2
    sys = mdlUpdate(t,x,u,dperiod,doffset); % Update disc states

  case 3
    sys = mdlOutputs(t,x,u,doffset,dperiod); % Calculate outputs

  case {4, 9}
    sys = [];       % Unused flags

  otherwise
    error(['unhandled flag = ',num2str(flag)]); % Error handling
end
% end of mixedm
%
%==============================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the
% S-function.
%==============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(dperiod,doffset)
sizes = simsizes;
sizes.NumContStates  = 1;
sizes.NumDiscStates  = 1;
sizes.NumOutputs     = 1;
sizes.NumInputs      = 1;
sizes.DirFeedthrough = 0;
sizes.NumSampleTimes = 2;
sys = simsizes(sizes);
x0  = ones(2,1);
str = [];
ts  = [0,       0           % sample time
       dperiod, doffset];
% end of mdlInitializeSizes
%
```

```
%=============================================================
% mdlDerivatives
% Compute derivatives for continuous states.
%=============================================================
%
function sys = mdlDerivatives(t,x,u,dperiod,doffset)
sys = u;
% end of mdlDerivatives
%
%=============================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time
% step requirements.
%=============================================================
%
function sys = mdlUpdate(t,x,u,dperiod,doffset)
% next discrete state is output of the integrator
% Return next discrete state if we have a sample hit within a
% tolerance of 1e-8. If we don't have a sample hit, return [] to
% indicate that the discrete state shouldn't change.
%
if abs(round((t-doffset)/dperiod)-(t-doffset)/dperiod) < 1e-8
  sys = x(1);
else
  sys = []; % This is not a sample hit, so return an empty
end         % matrix to indicate that the states have not changed.
% end of mdlUpdate
%
%=============================================================
% mdlOutputs
% Return the output vector for the S-function
%=============================================================
%
function sys = mdlOutputs(t,x,u,doffset,dperiod)
% Return output of the unit delay if we have a
% sample hit within a tolerance of 1e-8. If we
% don't have a sample hit then return [] indicating
% that the output shouldn't change.
%
if abs(round((t-doffset)/dperiod)-(t-doffset)/dperiod) < 1e-8
```

```
  sys = x(2);
else
  sys = []; % This is not a sample hit, so return an empty
end         % matrix to indicate that the output has not changed

% end of mdlOutputs
```

Example - Variable Step S-Functions

This M-file is an example of an S-function that uses a variable step time. This example, in an M-file called `vsfunc.m`, calls `mdlGetTimeOfNextVarHit` when `flag = 4`. Because the calculation of a next sample time depends on the input `u`, this block has direct feedthrough. Generally, all blocks that use the input to calculate the next sample time (`flag = 4`) require direct feedthrough. Here is the code for the M-file S-function:

```
function [sys,x0,str,ts] = vsfunc(t,x,u,flag)
% This example S-function illustrates how to create a variable
% step block in Simulink.  This block implements a variable step
% delay in which the first input is delayed by an amount of time
% determined by the second input:
%
%     dt      = u(2)
%     y(t+dt) = u(t)
%
switch flag,

  case 0
    [sys,x0,str,ts] = mdlInitializeSizes; % Initialization

  case 2
    sys = mdlUpdate(t,x,u); % Update Discrete states

  case 3
    sys = mdlOutputs(t,x,u); % Calculate outputs

  case 4
    sys = mdlGetTimeOfNextVarHit(t,x,u); % Get next sample time

  case { 1, 9 }
    sys = []; % Unused flags
```

```
  otherwise
    error(['Unhandled flag = ',num2str(flag)]); % Error handling
end
% end of vsfunc
%==============================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the
% S-function.
%==============================================================
%
function [sys,x0,str,ts] = mdlInitializeSizes
%
% call simsizes for a sizes structure, fill it in and convert it
% to a sizes array
%
sizes = simsizes;
sizes.NumContStates  = 0;
sizes.NumDiscStates  = 1;
sizes.NumOutputs     = 1;
sizes.NumInputs      = 2;
sizes.DirFeedthrough = 1;  % flag=4 requires direct feedthrough
                           % if input u is involved in
                           % calculating the next sample time
                           % hit.
sizes.NumSampleTimes = 1;
sys = simsizes(sizes);
%
% Initialize the initial conditions
%
x0 = [0];
%
% Set str to an empty matrix
%
str = [];
%
% Initialize the array of sample times
%
ts = [-2 0];      % variable sample time
% end of mdlInitializeSizes
%
```

```
%=============================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time
% step requirements.
%=============================================================
%
function sys = mdlUpdate(t,x,u)
sys = u(1);
% end of mdlUpdate
%
%=============================================================
% mdlOutputs
% Return the block outputs.
%=============================================================
%
function sys = mdlOutputs(t,x,u)
sys = x(1);
% end mdlOutputs
%
%=============================================================
% mdlGetTimeOfNextVarHit
% Return the time of the next hit for this block.  Note that the
% result is absolute time.
%=============================================================
%
function sys = mdlGetTimeOfNextVarHit(t,x,u)
sys = t + u(2);
% end of mdlGetTimeOfNextVarHit
```

`mdlGetTimeOfNextVarHit` returns the "time of the next hit," the time in the simulation when `vsfunc` is next called. This means that there is no output from this S-function until the time of the next hit. In `vsfunc`, the time of the next hit is set to `t + u(2)`, which means that the second input, `u(2)`, sets the time when the next call to `vsfunc` occurs.

Passing Additional Parameters

Simulink always passes `t`, `x`, and `u` into S-functions. It is possible to pass additional parameters into your M-file S-function. For an example of how to do this, see `limintm.m` in the `toolbox/simulink/blocks` directory.

Writing S-Functions as C MEX-Files

A C MEX-file that defines an S-Function block must provide information about the model; Simulink needs that information during simulation. As the simulation proceeds, Simulink, the ODE solver, and the MEX-file interact to perform specific tasks. These tasks include defining initial conditions and block characteristics, and computing derivatives, discrete states, and outputs.

C MEX-file S-functions have the same structure and perform the same functions as M-file S-functions. Simulink includes a template file for writing C MEX-file S-functions, called `sfuntmpl.c`.

The MEX-file must define specific functions that provide the information needed to perform these tasks, as well as contain statements that include other necessary code. The table below describes the functions that Simulink calls during the simulation, in the order they are called. Every S-function MEX-file must contain all of these functions, even if not all of them are used in a model. These functions are described in detail later in this section.

Table 8-3: Functions Called During a Simulation

Simulation Stage	S-Function Routine

Initialization of block size information, sample times, and initial conditions	`mdlInitializeSizes`
	`mdlInitializeSampleTimes`
	`mdlInitializeConditions`

Calculation of outputs	`mdlOutputs`
Update of discrete states	`mdlUpdate`
Calculation of next sample hit (optional)	`mdlGetTimeOfNextVarHit`
Calculation of derivatives	`mdlDerivatives`
Perform tasks at end of simulation	`mdlTerminate`

Unlike M-file S-functions, there is not an explicit `flag` parameter associated with each S-function routine. This is because Simulink automatically calls each S-function routine at the appropriate time in the simulation stage. Also, there are S-function routines associated with C MEX-file S-functions that don't have counterparts in M-file S-functions. These include `mdlInitializeSampleTimes` and `mdlInitializeConditions`.

Simulink maintains information about the S-Function block in a data structure called the `SimStruct`. The `include` file that defines the `SimStruct` provides macros that enable your MEX-file to set values in and get values from the `SimStruct`. The statement that includes the definition of the `SimStruct` is shown in “Statements Required at the Top of the File” on page 8–27. The macros that access the `SimStruct` are listed in Appendix C.

Simulink provides a template C MEX-file S-function that includes the statements that define the necessary functions, as well as comments that should help you write the code needed for your S-function block. This template file, `sfuntmpl.c`, can be found in the directory `simulink/src` below the MATLAB root directory.

NOTE We recommend that you use the C MEX-file template when developing MEX S-functions.

Statements Required at the Top of the File

The MEX-file must contain a statement that includes the definition of the `SimStruct` data structure that stores pointers to the data used by the simulation. The included code also defines the macros used to store and retrieve data in the `SimStruct`, described in detail in Appendix C. This statement must precede the function definitions.

```
#include "simstruc.h"
```

Define the name of your S-function with a statement like the one below, substituting your model name for *your_sfunction_name_here*:

```
#define S_FUNCTION_NAME your_sfunction_name_here
```

Statements Required at the Bottom of the File

Include this code at the end of your C MEX-file S-function:

```
#ifdef MATLAB_MEX_FILE   /* Is this being compiled as MEX-file? */
#include "simulink.c"    /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"     /* Code generation registration func */
#endif
```

These statements select the appropriate code for your particular application:

- `simulink.c` is included if the file is being compiled into a MEX-file.
- `cg_sfun.h` is included if the file is being used in conjunction with the Simulink Real-Time Workshop to produce a stand-alone or real-time executable.

Defining S-Function Block Characteristics

The `sizes` structure in the `SimStruct` stores essential size information about the S-Function block, including the number of inputs, outputs, states, and other block characteristics. The `sizes` structure is initialized in the `mdlInitializeSizes` function. Supplied macros set values for the structure fields. If a value is not specified, it is initialized to zero.

The table below describes the fields in the `sizes` structure and indicates the macros used to define values for the fields:

Table 8-4: sizes Structure Fields

Structure Field	Macro that Sets Value
Number of continuous states	`ssSetNumContStates(S,numContStates)`
Number of discrete states	`ssSetNumDiscStates(S,numDiscStates)`
Number of outputs	`ssSetNumOutputs(S,numOutputs)`
Number of inputs	`ssSetNumInputs(S,numInputs)`
Flag for direct feedthrough	`ssSetDirectFeedThrough(S,dirFeedThru)`
Number of sample times	`ssSetNumSampleTimes(S,numSampleTimes)`
Number of input arguments	`ssSetNumInputArgs(S,numInputArgs)`
Number of integer work vector elements	`ssSetNumIWork(S,numIWork)`

Table 8-4: sizes Structure Fields (Continued)

Structure Field	Macro that Sets Value
Number of real work vector elements	`ssSetNumRWork(S,numRWork)`
Number of pointer work vector elements	`ssSetNumPWork(S,numPWork)`

There are additional macros that get various values. For a complete list of built-in macros that work with C language S-functions, see Appendix C.

Setting Values for sizes Structure Fields

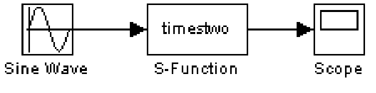
Each macro call has two arguments: the first is the `SimStruct` structure identifier, `S`, and the second is the data value. For example, the following statements define the `sizes` structure for an S-function:

```
static void mdlInitializeSizes(SimStruct *S)
{
  ssSetNumContStates(S, 1);
  ssSetNumDiscStates(S, 0);
  ssSetNumOutputs(S, 1);
  ssSetNumInputs(S, 1);
  ssSetDirectFeedThrough(S, 0);
  ssSetNumSampleTimes(S, 1);
}
```

This code specifies that the block has one continuous state and no discrete states, one output and one input, no direct feedthrough, and one sample time (a continuous block must still define a sample time of zero).

A Simple C MEX-File Example

“Writing S-Functions as M-Files” on page 8–11 started with a simple example that doubled the amplitude of an input signal. The example below does the same thing, except that it is written in C MEX-file format:



The C code that contains the S-function is modeled on a Simulink S-function template called `sfuntmpl.c`. Using this template, you can create a C function by following the structure provided. Below is the MATLAB code in `timestwo.c`, the C-code that contains the S-function:

```
/*
 * TIMESTWO An example C S-function for multiplying an input by 2
 *
 *       y = 2*x
 *
 */
#define S_FUNCTION_NAME timestwo
#include "simstruc.h"

/* mdlInitializeSizes - initialize the sizes structure */

static void mdlInitializeSizes(SimStruct *S)
{
  ssSetNumContStates( S, 0);   /* number of continuous states */
  ssSetNumDiscStates(S,0);     /* number of discrete states */
  ssSetNumInputs(S,DYNAMICALLY_SIZED); /* number of inputs */
  ssSetNumOutputs(S,DYNAMICALLY_SIZED);/* number of outputs */
  ssSetDirectFeedThrough(S,1);   /* direct feedthrough flag */
  ssSetNumSampleTimes(S,1);      /* number of sample times */
  ssSetNumSFcnParams(S,0); /* number of extra input param's */
  ssSetNumRWork(S,0); /* number of real work vector elements */
  ssSetNumIWork(S,0); /* number of integer work vector elem's*/
  ssSetNumPWork(S,0); /* number of pointer work vector elem's*/
}

/* mdlInitializeSampleTimes - initialize sample times array */
static void mdlInitializeSampleTimes(SimStruct *S)
{
  ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
  ssSetOffsetTime(S, 0, 0.0);
}

/* mdlInitializeConditions - initialize the states */

static void mdlInitializeConditions(double *x0, SimStruct *S)
```

```
{
}

/* mdlOutputs - compute the outputs */

static void mdlOutputs(double *y, const double *x,
                       const double *u, SimStruct *S, int tid)
{
  int i, nOutputs;
  nOutputs = ssGetNumOutputs(S);
  for (i=0; i<nOutputs; i++){
    *y++ = 2.0*(*u++);
  }
}

/* mdlUpdate - perform action at major integration time step */

static void mdlUpdate(double *x, const double *u,
                      SimStruct *S, int tid)
{
}

/* mdlDerivatives - compute the derivatives */

static void mdlDerivatives(double *dx, const double *x,
                           const double *u,SimStruct *S,int tid)
{
}

/* mdlTerminate - called when the simulation is terminated */

static void mdlTerminate(SimStruct *S)
{
}
#ifdef MATLAB_MEX_FILE   /* Is this being compiled as MEX-file? */
#include "simulink.c"    /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"     /* Code generation registration func */
#endif
```

An important distinction between M-file and C MEX-file S-functions is that C MEX-file S-functions must have calls to `mdlUpdate` and `mdlDerivative`, even if there are no continuous or discrete states in the model. Without these functions, your MEX-file will not compile successfully.

Examples of C MEX-File S-Function Blocks

In general, most S-Function blocks require the handling of states, continuous or discrete. The following sections discuss four common types of systems that you can model in Simulink with S-functions:

- Continuous
- Discrete
- Hybrid
- Variable step time

All examples are based on the C MEX-file S-function template, `sfuntmpl.c`.

Example - Continuous State S-Function

This example shows how to model a set of continuous state equations in C. The example is called `csfunc.c` and is located in the `simulink/src` directory:

```
/*
 * CSFUNC An example C-file S-function for defining a continuous
 * system.
 *      x' = Ax + Bu
 *      y  = Cx + Du
 */
#define S_FUNCTION_NAME csfunc
#include "simstruc.h"
static double A[2][2]={ { -0.09, -0.01 } ,
                        {  1   ,  0    }
                      };
static double B[2][2]={ {  1   , -7    } ,
                        {  0   , -2    }
                      };
static double C[2][2]={ {  0   ,  2    } ,
                        {  1   , -5    }
                      };
```

```
static double D[2][2]={ { -3   ,  0    } ,
                        {  1   ,  0    }
                      };

/* mdlInitializeSizes - initialize the sizes structure */

static void mdlInitializeSizes(SimStruct *S)
{
  ssSetNumContStates(S,2); /* number of continuous states */
  ssSetNumDiscStates(S,0); /* number of discrete states */
  ssSetNumInputs(S,2);     /* number of inputs */
  ssSetNumOutputs(S,2);    /* number of outputs */
  ssSetDirectFeedThrough(S,1); /* direct feedthrough flag
                                  Matrix D is nonempty. */
  ssSetNumSampleTimes(S,1); /* number of sample times */
  ssSetNumSFcnParams(S,0);  /* number of input arguments */
  ssSetNumRWork(S,0); /* number of real work vector elements */
  ssSetNumIWork(S,0); /* number of integer work vector elem's*/
  ssSetNumPWork(S,0); /* number of pointer work vector elem's*/
}

/* mdlInitializeSampleTimes - initialize sample times array */

static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}


/* mdlInitializeConditions - initialize the states */

static void mdlInitializeConditions(double *x0, SimStruct *S)
{
  int i;
  for (i=0; i<2; i++){
    *x0++ = 0.0;
  }
}
/* mdlOutputs - compute the outputs */
```

```
static void mdlOutputs(double *y, const double *x,
                       const double *u, SimStruct *S, int tid)
{
  /* y = Cx+Du */
  y[0] = C[0][0]*x[0]+C[0][1]*x[1]+D[0][0]*u[0]+D[0][1]*u[1];
  y[1] = C[1][0]*x[0]+C[1][1]*x[1]+D[1][0]*u[0]+D[1][1]*u[1];
}

/* mdlUpdate - perform action at major integration time step */

static void mdlUpdate(double *x, const double *u, SimStruct *S,
                      int tid)
{
}

/* mdlDerivatives - compute the derivatives */

static void mdlDerivatives(double *dx, contst double *x,
                           const double *u,SimStruct *S,int tid)
{
  /* xdot = Ax+Bu */
  dx[0] = A[0][0]*x[0]+A[0][1]*x[1]+B[0][0]*u[0]+B[0][1]*u[1];
  dx[1] = A[1][0]*x[0]+A[1][1]*x[1]+B[1][0]*u[0]+B[1][1]*u[1];
}
/* mdlTerminate - called when the simulation is terminated */

static void mdlTerminate(SimStruct *S)
{
}

#ifdef MATLAB_MEX_FILE   /* Is this being compiled as MEX-file? */
#include "simulink.c"    /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"     /* Code generation registration func */
#endif
```

This MEX-file is the C language version of `csfunc.m`, the M-file code that implements a continuous state model. The derivative of the state vector `x` is calculated in `mdlDerivative`, and the output `y` is calculated in `mdlOutputs`.

`mdlUpdate` and `mdlTerminate` are empty functions because there are no discrete states or tasks to complete at termination.

Example - Discrete State S-Function

Simulink includes a function called `dsfunc.c`, which is an example of a discrete state system modeled in an S-function. Here is the code for the C language S-function:

```
 /* DSFUNC An example C-file S-function for defining a discrete
  * system.
  *       x(n+1) = Ax(n) + Bu(n)
  *       y(n)   = Cx(n) + Du(n)
  */

#define S_FUNCTION_NAME dsfunc
#include "simstruc.h"
static double A[2][2]={ { -1.3839, -0.5097 } ,
                        {  1     ,  0      }
                      };
static double B[2][2]={ { -2.5559,  0      } ,
                        {  0     ,  4.2382 }
                      };
static double C[2][2]={ {  0     ,  2.0761 } ,
                        {  0     ,  7.7891 }
                      };
static double D[2][2]={ { -0.8141, -2.9334 } ,
                        {  1.2426,  0      }
                      };
/* mdlInitializeSizes - initialize the sizes structure */
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumContStates(S,0); /* number of continuous states */
    ssSetNumDiscStates(S,2); /* number of discrete states */
    ssSetNumInputs(S,2);          /* number of inputs */
    ssSetNumOutputs(S,2);         /* number of outputs */
    ssSetDirectFeedThrough(S,1);  /* direct feedthrough flag
                                     Matrix D is nonempty */
    ssSetNumSampleTimes(S,1);     /* number of sample times */
    ssSetNumSFcnParams(S,0);   /* number of input arguments */
    ssSetNumRWork(S,0); /* number of real work vector elements*/
```

```
    ssSetNumIWork(S,0); /* number of integer work vector elems*/
    ssSetNumPWork(S,0); /* number of pointer work vector elems*/
}
/* mdlInitializeSampleTimes - initialize sample times array */

static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, 1.0);
    ssSetOffsetTime(S, 0, 0.0);
}

/* mdlInitializeConditions - initialize the states */

static void mdlInitializeConditions(double *x0, SimStruct *S)
{
  int i;
  for (i=0; i<2; i++){
    *x0++ = 1.0;
  }
}

/* mdlOutputs - compute the outputs */

static void mdlOutputs(double *y, const double *x,
                       const double *u, SimStruct *S, int tid)
{
  /* y = Cx+Du */
  y[0] = C[0][0]*x[0]+C[0][1]*x[1]+D[0][0]*u[0]+D[0][1]*u[1];
  y[1] = C[1][0]*x[0]+C[1][1]*x[1]+D[1][0]*u[0]+D[1][1]*u[1];
}

/* mdlUpdate - perform action at major integration time step */

static void mdlUpdate(double *x, const double *u, SimStruct *S,
                      int tid)
{
  double tempX[2];
  /* xdot = Ax+Bu */
  tempX[0]= A[0][0]*x[0]+A[0][1]*x[1]+B[0][0]*u[0]+B[0][1]*u[1];
  tempX[1]= A[1][0]*x[0]+A[1][1]*x[1]+B[1][0]*u[0]+B[1][1]*u[1];
```

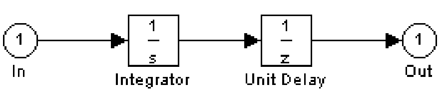
```
  x[0] = tempX[0];
  x[1] = tempX[1];
}
/* mdlDerivatives - compute the derivatives */

static void mdlDerivatives(double *dx, const double *x,
                           const double *u,SimStruct *S,int tid)
{
}

/* mdlTerminate - called when the simulation is terminated */

static void mdlTerminate(SimStruct *S)
{
}

#ifdef MATLAB_MEX_FILE /* Is this being compiled as a MEX-file? */
#include "simulink.c"  /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"   /* Code generation registration func */
#endif
```

This is the C language equivalent of `dsfunc.m`. The updating of discrete state vector `x` occurs in `mdlUpdate`, and the calculation of the outputs occurs in `mdlOutputs`. Since there are no continuous states or tasks to complete at termination, both `mdlDerivatives` and `mdlTerminate` are empty functions.

Example - Hybrid System S-Functions

Simulink includes a function called `mixedm.c`, which is an example of a hybrid (a combination of continuous and discrete states) system modeled in an S-function. `mixedm.c` combines elements of `csfunc.c` and `dsfunc.c`. If you have a hybrid model, all you have to do is place your continuous equations in `mdlDerivative` and your discrete equations in `mdlUpdate`.

In Simulink block diagram form, the model looks like this:



Here is the code for the C language S-function `mixedm.c`, which implements a continuous integrator followed by a discrete unit delay:

```
 /* mixedm, a C-file S-function that models a continuous
  * integrator (1/s) in series with a unit delay (1/z).
  */
#define S_FUNCTION_NAME mixedm
#include "simstruc.h"

/* mdlInitializeSizes - initialize the sizes structure */

static void mdlInitializeSizes(SimStruct *S)
{
 ssSetNumContStates(S,1); /* number of continuous states */
 ssSetNumDiscStates(S,1); /* number of discrete states */
 ssSetNumInputs(S,1);     /* number of inputs */
 ssSetNumOutputs(S,1);    /* number of outputs */
 ssSetDirectFeedThrough(S,0);   /* direct feedthrough flag */
 ssSetNumSampleTimes(S,2);   /* number of sample times */
 ssSetNumSFcnParams(S,0);   /* number of input arguments */
 ssSetNumRWork(S,0); /* number of real work vector elem's */
 ssSetNumIWork(S,0); /* number of integer work vector elems*/
 ssSetNumPWork(S,0); /* number of pointer work vector elems*/
}
/* mdlInitializeSampleTimes - initialize sample times array */

static void mdlInitializeSampleTimes(SimStruct *S)
{
 ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
 ssSetOffsetTime(S, 0, 0.0);
 ssSetSampleTime(S, 1, 1.0);
 ssSetOffsetTime(S, 1, 0.0);
}
/* mdlInitializeConditions - initialize the states */

static void mdlInitializeConditions(double *x0, SimStruct *S)
{
 int i;
 for (i=0; i<2; i++){
    *x0++ = 0.0;
```

```
 }
}

/* mdlOutputs - compute the outputs */

static void mdlOutputs(double *y, const double *x,
                       const double *u, SimStruct *S, int tid)
{
  if (ssIsSampleHit(S,1,tid)) {
    y[0] = x[1]; /* The discrete output of the S-function. */
  }
}
/* mdlUpdate - perform action at major integration time step */

static void mdlUpdate(double *x, const double *u, SimStruct *S,
                      int tid)
{
  if (ssIsSampleHit(S,1,tid)) {
   x[1] = x[0]; /* The discrete state of the S-function. */
  }
}
/* mdlDerivatives - compute the derivatives */

static void mdlDerivatives(double *dx, const double *x,
                           const double *u,SimStruct *S,int tid)
{
  dx[0] = u[0];
}
/* mdlTerminate - called when the simulation is terminated */

static void mdlTerminate(SimStruct *S)
{
}
#ifdef MATLAB_MEX_FILE   /* Is this being compiled as MEX-file? */
#include "simulink.c"    /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"     /* Code generation registration func */
#endif
```

Since there are no tasks to complete at termination, `mdlTerminate` is an empty function. `mdlDerivatives` calculates the derivatives of the continuous states of the state vector `x`, and `mdlUpdate` contains the equations used to update the discrete states of `x`.

Example - Variable Step S-Functions

This example is of an S-function that uses a variable step time. Variable step-size functions require a call to `mdlGetTimeOfNextVarHit`, which is an S-function routine that calculates the time of the next sample hit.

The S-function is in an C program called `vsfunc.c` located in your Simulink directory. `vsfunc` is a discrete S-function that delays its first input by an amount of time determined by the second input. Here is the C code for `vsfunc.c`:

```
/* vsfunc Variable step S-function example.
 * This block implements a variable step delay in which the first
 * input is delayed by an amount of time determined by the second
 * input:
 *
 *     dt      = u(2)
 *     y(t+dt) = u(t)
 */

#define S_FUNCTION_NAME vsfunc
#include "simstruc.h"

/* mdlInitializeSizes - initialize the sizes structure */

static void mdlInitializeSizes(SimStruct *S)
{
 ssSetNumContStates(S,0);   /* number of continuous states */
 ssSetNumDiscStates(S,1);   /* number of discrete states */
 ssSetNumInputs(S,2);       /* number of inputs */
 ssSetNumOutputs(S,1);      /* number of outputs */
 ssSetDirectFeedThrough(S,1);   /* direct feedthrough flag -
                                 * set because next sample hit
                                 * depends on input u. */
 ssSetNumSampleTimes(S,1);  /* number of sample times */
 ssSetNumSFcnParams(S,0);   /* number of input arguments */
```

```
 ssSetNumRWork(S,0); /* number of real work vector elements */
 ssSetNumIWork(S,0); /* number of integer work vector elem's*/
 ssSetNumPWork(S,0); /* number of pointer work vector elem's*/
}

/* mdlInitializeSampleTimes - initialize sample times array */

static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, VARIABLE_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

/* mdlInitializeConditions - initialize the states */

static void mdlInitializeConditions(double *x0, SimStruct *S)
{
   x0[0] = 0.0;
}

/* mdlGetTimeOfNextVarHit - Get the time of the next
 * variable sample time hit */

#define MDL_GET_TIME_OF_NEXT_VAR_HIT
static void mdlGetTimeOfNextVarHit(SimStruct *S)
{
  double *u = ssGetU(S);
  ssSetTNext(S, ssGetT(S)+u[1]);
}

/* mdlOutputs - compute the outputs */

static void mdlOutputs(double *y, const double *x,
                       const double *u, SimStruct *S, int tid)
{
   y[0] = x[0];
}

/* mdlUpdate - perform action at all time steps */
```

```
static void mdlUpdate(double *x, const double *u, SimStruct *S,
                      int tid)
{
   x[0] = u[0];
}
/* mdlDerivatives - compute the derivatives */

static void mdlDerivatives(double *dx, const double *x,
                           const double *u,SimStruct *S,int tid)
{
}

/* mdlTerminate - called when the simulation is terminated */

static void mdlTerminate(SimStruct *S)
{
}
#ifdef MATLAB_MEX_FILE   /* Is this being compiled as MEX-file? */
#include "simulink.c"    /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"     /* Code generation registration func */
#endif
```

The output of vsfunc is simply the input u delayed by a variable amount of time. mdlOutputs sets the output y equal to state x. mdlUpdate sets the state vector x equal to u, the input vector. This example calls mdlGetTimeOfNextVarHit, an S-function routine that calculates and sets the "time of next hit," that is, the time when is vsfunc is next called. In mdlGetTimeOfNextVarHit the macro ssGetU is used to get a pointer to the input u. Then this call is made:

```
ssSetTNext(S, ssGetT(S)+u[1]);
```

The macro ssGetT gets the simulation time t. The second input to the block, u[1], is added to t, and the macro ssSetTNext sets the time of next hit equal to t+u[1], delaying the output by the amount of time set in u[1].

Creating General Purpose S-Function Blocks

General purpose S-Function blocks can be used in a variety of applications. To make the block generic, you can specify that certain block characteristics be

dependent on the number of inputs to the S-Function block. To do this, assign the macro `DYNAMICALLY_SIZED` to the dependent characteristics; the value of these `sizes` fields are determined during the compilation of the simulation. The `sizes` structure fields that can be set based on the number of inputs include:

- The number of continuous states
- The number of discrete states
- The number of outputs
- The number of inputs
- The number of real work vector elements
- The number of integer work vector elements
- The number of pointer work vector elements

For a discussion of work vectors, see "Allocating Work Vectors and Setting Their Values" on page 8–48.

The following statements, included in the `mdlInitializeSizes` function, set the number of continuous states, the number of outputs, and the number of inputs to the number of S-Function block inputs:

```
ssSetNumContStates(S, DYNAMICALLY_SIZED);
ssSetNumOutputs(S, DYNAMICALLY_SIZED);
ssSetNumInputs(S, DYNAMICALLY_SIZED);
```

Note that dynamically sized S-functions can determine when they aren't connected using the `ssGetInputConnected(S)` and `ssGetOutputConnected(S)` macros.

Specifying Parameter Values Interactively

In addition to their inputs, outputs, and states, S-functions can also accept parameters. These parameters are set interactively using the **S-Function parameters** field of the block's dialog box. If you define parameters interactively, you should follow these steps when you create the S-function:

1. Determine the order in which the parameters are to be specified in the block's dialog box.
2. In the `mdlInitializeSizes` function, use the `ssSetNumInputArgs` macro to tell Simulink how many parameters are passed in to the S-function. Specify

S as the first argument and the number of parameters you are defining interactively as the second argument.

3 Access these input arguments in the S-function using the ssGetArg macro. Specify S as the first argument and the relative position of the parameter in the list entered on the dialog box (0 is the first position) as the second argument.

When you run the simulation, specify parameter names or values in the **S-Function parameters** field of the block's dialog box. The order of the parameters names or values must be identical to the order that you defined them in step 1 above. If you specify variable names, they do not need to be the same as the names used in the MEX-file.

For example, the following code is part of a device driver S-function. Four input parameters are used: BASE_ADDRESS_ARG, GAIN_RANGE_ARG, PROG_GAIN_ARG, and NUM_OF_CHANNELS_ARG. The code uses #define statements to associate particular input arguments with the parameter names.

```
/* Input Arguments */
#define BASE_ADDRESS_ARG     ssGetArg(S, 0)
#define GAIN_RANGE_ARG       ssGetArg(S, 1)
#define PROG_GAIN_ARG        ssGetArg(S, 2)
#define NUM_OF_CHANNELS_ARG  ssGetArg(S, 3)
```

When running the simulation, the user enters four variable names or values in the **S-Function parameters** field of the block's dialog box. The first corresponds to the first expected parameter, BASE_ADDRESS_ARG. The second corresponds to the next expected parameter, and so on.

The mdlInitializeSizes function contains this statement:

```
ssSetNumInputArgs(S, 4);
```

Simulink always passes t, x, and u into S-functions. These parameters are not included in **S-Function parameters**. Only additional parameters that you want to pass to the S-function are included here. For an example to how to pass additional parameters, see dlimint.c in the simulink/src directory.

Parameter Changes

To notify an S-function of parameter changes, the S-function must register a mdlCheckParameters routine. This routine will be called any time after mdlInitializeSizes has been called.

When Simulink Calls the mdlUpdate Function

An S-function's `mdlUpdate` function is called if and only if the S-function has one or more discrete states or does *not* have direct feedthrough. The reason for this is that most S-functions that do not have discrete states and do have direct feedthrough do not have update functions. Therefore, Simulink is able to eliminate the need for the extra call in these circumstances.

If your S-function needs to have its `mdlUpdate` routine called and it does not satisfy either of the above conditions, specify that it has a discrete state using the `ssSetNumDiscStates` macro in the `mdlInitializeSizes` function.

Specifying Sample Times

The sample times are stored in an array in the `SimStruct` and specified in the `mdlInitializeSampleTimes` function. The value is assigned by the `ssSetSampleTime` macro, which has this syntax:

```
ssSetSampleTime(S, st_index, sample_time);
```

where `S` is the `SimStruct`, `st_index` is the index of the sample time being assigned a value (the first is 0, the second is 1, and so on), and `sample_time` is the value associated with the particular `st_index`.

The `mdlInitializeSizes` function must include a call to the macro that specifies the number of sample times defined for the block (`num`):

```
ssSetNumSampleTimes(S, num);
```

Sample Times and Input/Output Port Widths

Sample times can be a function of the input/output port widths. In `mdlInitializeSampleTimes`, you can specify that sample times are a function of `ssGetNumInputs` and `ssGetNumOutputs`.

Specifying Offsets

The offsets are also stored in a `SimStruct` array and specified in the `mdlInitializeSampleTimes` function. The `ssSetOffsetTime` macro, which uses this calling syntax, assigns the value:

```
ssSetOffsetTime(S, st_index, offset_time);
```

where S is the SimStruct, st_index is the index of the sample time to which the offset applies, and offset_time is the value associated with the specified offset.

Multirate S-Function Blocks

In a multirate S-Function block, encapsulate the code that defines each behavior in the mdlOutput and mdlUpdate functions with a statement that determines whether a sample hit has occurred at the current time value. The ssIsSampleHit macro determines whether the current time is a sample hit for a specified sample time. The macro has this syntax:

```
ssIsSampleHit(S, st_index, tid)
```

where S is the SimStruct, st_index identifies a specific sample time index, and tid is the task ID (tid is an argument to the mdlOutput, mdlUpdate, and mdlDerivatives functions).

For example, these statements specify three sample times: one for continuous behavior, and two for discrete behavior.

```
ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
ssSetSampleTime(S, 1, 0.75);
ssSetSampleTime(S, 2, 1.0);
```

In the mdlUpdate function, the following statement would encapsulate the code that defines the behavior for the sample time of 0.75 seconds:

```
if (ssIsSampleHit(S, 1, tid)) {
}
```

The second argument, 1, corresponds to the second sample time, 0.75 seconds.

Example - Defining a Sample Time for a Continuous Block

This example defines a sample time for a block that is continuous in nature.

```
/* Initialize the sample time and offset */
static void mdlInitializeSampleTimes(SimStruct *S)
{
  ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
  ssSetOffsetTime(S, 0, 0.0);
}
```

The following statement also appears in the `mdlInitializeSizes` function:

```
ssSetNumSampleTimes(S, 1);
```

Example - Defining a Sample Time for a Hybrid Block

This example defines sample times for a hybrid S-Function block.

```
/* Initialize the sample time and offset */
static void mdlInitializeSampleTimes(SimStruct *S)
{
  /* Continuous state sample time and offset */
  ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
  ssSetOffsetTime(S, 0, 0.0);

  /* Discrete state sample time and offset */
  ssSetSampleTime(S, 1, 0.1);
  ssSetOffsetTime(S, 1, 0.025);
}
```

The offset causes Simulink to call the `mdlUpdate` function at these times: 0.025 seconds, 0.125 seconds, 0.225 seconds, and so on, in increments of 0.1 seconds.

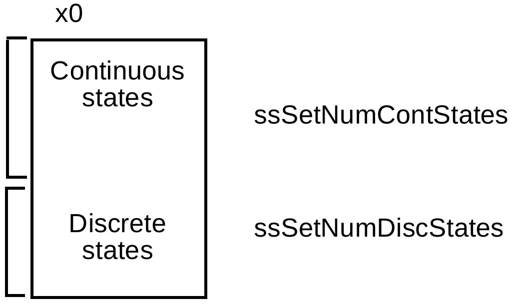
The following statement, which indicates how many sample times are defined, also appears in the `mdlInitializeSizes` function:

```
ssSetNumSampleTimes(S, 2);
```

Setting the Initial Conditions

The initial conditions for the S-Function block's states are defined in the `mdlInitializeConditions` function. The `x0` vector contains the initial conditions. Simulink calls `mdlInitializeConditions` once during the simulation to set the initial conditions. When the S-function is within an enabled subsystem, the `mdlInitializeConditions` is called each time that the subsystem becomes enabled.

The initial conditions vector is represented by this diagram, which shows the order in which the states are defined and indicates the macros used to define the number of states.



Example - Defining a Vector of Initial Conditions

This example shows how to define a vector of initial conditions. The following code initializes the continuous states to 1.0.

```
/* Set initial conditions */
static void mdlInitializeConditions(double *x0,SimStruct *S)
{
  int i, nCStates;

  nCStates = ssGetNumContStates(S);
  for (i = 0; i < nCStates; i++)
    *x0++ = 1.0;
}
```

Allocating Work Vectors and Setting Their Values

If your S-function needs persistent memory storage, use S-function *work vectors* instead of static or global variables. If you use static or global variables, they are used by multiple instances of your S-function. The ability to keep track of multiple instances of an S-function is called *reentrancy*.

You can create an S-function that is reentrant by using work vectors. These are persistent storage locations that Simulink manages for an S-function. Integer, floating point, and pointer data types are supported. The number of elements in each vector can be specified dynamically as a function of the number of inputs to the S-function.

Macros Used with Work Vectors. Simulink provides macros for allocating the work vector elements, assigning variables to the elements, and accessing these variables.

- Include in the `mdlInitializeSizes` function macros that define the number of work vector elements:

  `ssSetNumRWork(S,numRWork)` defines the number of real work vector elements.

  `ssSetNumIWork(S,numIWork)` defines the number of integer work vector elements.

  `ssSetNumPWork(S,numPWork)` defines the number of pointer work vector elements.

- Include in the `mdlInitializeConditions` function macros that set the work vector element values:

  `ssSetRWorkValue(S,rworkIdx,rworkValue)` defines the value of real work vector element `rworkIdx` to be `rworkValue`.

  `ssSetIWorkValue(S,iworkIdx,iworkValue)` defines the value of integer work vector element `iworkIdx` to be `iworkValue`.

  `ssSetPWorkValue(S,pworkIdx,pworkValue)` defines the value of pointer work vector element `pworkIdx` to be `pworkValue`.

- These macros get the work vector element values:

  `ssGetRWorkValue(S,rworkIdx)` obtains the value of real work vector element `rworkIdx`.

  `ssGetIWorkValue(S,iworkIdx)` obtains the value of integer work vector element `iworkIdx`.

  `ssGetPWorkValue(S,pworkIdx)` obtains the value of pointer work vector element `pworkIdx`.

You specify the number of work vector elements your block uses in the `mdlInitializeSizes` function. You allocate memory referenced by pointer work vectors in the `mdlInitializeConditions` function and free the memory in the `mdlTerminate` function.

An Example Involving a Pointer Work Vector. This example opens a file and stores the FILE pointer in the pointer work vector.

The statement below, included in the `mdlInititalizeSizes` function, indicates that the pointer work vector is to contain one element:

```
ssSetNumPWork(S, 1)   /* pointer work vector */
```

The code below uses the pointer work vector to store a FILE pointer, returned from the standard I/O function, `fopen`:

```
static void mdlInitilzeConditions(double *x0, SimStruct #S)
{
  void *PWork = ssGetPWork(S);
  FILE *fPtr;
  fPtr = fopen("file.data", "r");
  PWork[0] = fPtr;
}
```

This code retrieves the FILE pointer from the pointer work vector and passes it to `fclose` to close the file:

```
static void mdlTerminate(SimStruct *S)
{
  void *PWork = ssGetPWork(S);
  FILE *fPtr;

  fPtr = PWork[0];
  if (fPtr != NULL)
    fclose(fPtr);

  PWork[0] = NULL;
}
```

Nonsampled Zero Crossings for Continuous S-Functions

Continuous S-functions can register nonsampled zero crossings, which are used to "hone-in" on state events (i.e., discontinuities in the first derivative) of some signal, usually a function of an input to your S-function. To register nonsampled zero crossings, set the number of nonsampled zero crossings in `mdlInitializeSizes` using:

```
ssSetNumNonsampledZCs(S, num)
```

Then, define the `mdlZeroCrossings` routine to return the nonsampled zero crossings. `sfun_zc.c` shows how to use nonsampled zero crossings.

Mode Work Vector

Simulink supports a work vector referred to as the *mode vector*. Elements are integer values that are typically used with nonsampled zero crossings. Specify the number of mode vector elements in `mdlInitializeSizes` using `ssSetNumModes(S,num)`. You can then access the mode vector using `ssGetModeVector`.

Output and Work Vector Widths

For S-functions with dynamically sized inputs and outputs, you can configure your output width to be a function of your input width and vice versa. To do this, define these routines:

```
int mdlGetInputPortWidth(int outputWidth) /* return input width */
int mdlGetOutputPortWidth(int inputWidth) /* return output width */
```

Also, you can configure your work vector widths based on the input width, output width, or sample times. To do this, define a `mdlSetWorkWidths` routine.

`sfun_dynsize.c` demonstrates how to use the `mdlSetInputPortWidth`, `mdlSetOutputPortWidth`, and `mdlSetWorkWidths` optional methods to configure the input port width, output port width, and real work vector length based on the size of the signal driving the S-function.

Removing Ports When No Inputs and/or Outputs

Any C MEX, Fortran MEX, or M-file S-function that indicates it has no input and/or outputs will have the corresponding port(s) removed from the S-Function block icon. The port is removed when the simulation starts or when you choose **Update Diagram** from the **Edit** menu.

Using Function-Call Subsystems

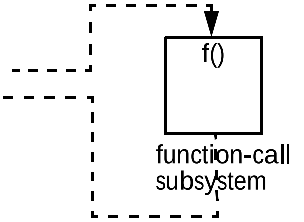
You can create a triggered subsystem whose execution is determined by logic internal to an S-function instead of by the value of a signal. A subsystem so

configured is called a *function-call subsystem*. To implement a function-call subsystems:

- In the Trigger block, select **function-call** as the **Trigger type** parameter.
- In the S-function, use the `ssCallSystem` macro to call the triggered subsystem.
- In the model, connect the S-Function block output directly to the trigger port.

Function-call subsystems are not executed directly by Simulink. The S-function determines when to execute the subsystem. When the subsystem completes execution, control returns to the S-function. This figure illustrates the interaction between a function-call subsystem and an S-function:

```
mdlOutputs()
    ...
    ssCallSystem(S,outputElement)
    next statement
    ...
```



Function-call subsystems can only be connected to S-functions that have been properly configured to accept them. To configure an S-function to call a function-call subsystem:

1. Specify which elements are to execute the function-call system in `mdlInitializeSampleTimes`. For example:

```
ssSetCallSystemOutput(S,0);  /* call on 1st element */
ssSetCallSystemOutput(S,2);  /* call on 3rd element */
```

2. Execute the subsystem in the appropriate `mdlOutputs`, `mdlUpdates`, or `mdlDerivatives` routine. For example:

```
static void mdlOutputs(...)
{
    if (((int)u[0]) % 2 == 1) {
        ssCallSystem(S,0);
    } else {
        ssCallSystem(S,2);
    }
...
}
```

`sfun_fcncall.c` illustrates an S-function configured to execute function-call subsystems.

Function-call subsystems are generally used by Stateflow™ blocks. For more information on their use, see the Stateflow documentation.

Instantaneous Update of S-Function Inputs

You can get instantaneous updates of S-function inputs by accessing the inputs through pointers. This is required for execution of function-call subsystems whose inputs feed back into your S-function. To configure your S-function to use input pointers, set, in `mdlInitializeSizes`:

```
ssSetOptions(S,SS_OPTION_USING_ssGetUPtrs)
```

Then, in all the routines where you access the input, include

```
UPtrsType uPtrs = ssGetUPtrs(S);
```

The `i`th input is then `*uPtrs[i]`.

Exception Handling

Each time an S-function is invoked, Simulink performs overhead tasks associated with exception handling. If the S-function does not contain any routines that can generate exceptions, you can improve the performance of the simulation. If you are not using routines that can throw an exception, set this option in `mdlInitializeSizes`:

```
ssSetOption(S, SS_OPTION_EXCEPTION_FREE_CODE);
```

Do not specify the `SS_OPTION_EXCEPTION_FREE_CODE` option if the S-function contains any routines that can throw an exception:

- All routines that start with `mex` can throw an exception.
- Routines that start with `mx` can throw an exception, except routines that get a pointer or determine the size of parameters, such as `mxGetPr`, `mxGetData`, `mxGetNumberOfDimensions`, `mxGetM`, `mxGetN`, and `mxGetNumberOfElements`.

If the S-function must allocate memory, avoid using `mxCalloc`. Instead, use the `stdlib.h` `calloc` routine directly and perform your own error handling, then free the memory in `mdlTerminate`.

Error Handling

Error handling works consistently across the Real-Time Workshop and Simulink. To report an error, S-functions should include these statements:

```
ssSetErrorStatus(S, "Error string");
return;
```

It is important that `Error string` be persistent memory, not a stack variable.

`sfun_errhdl.c` shows how to perform error checking. The S-function checks to see that required parameters are of the correct format.

Normal or Real-Time Workshop Simulation

S-functions can determine if they are running in a normal simulation, as part of Real-Time Workshop code generation, or as part of external mode. This is done using `ssGetSimMode(S)`.

Additional Macros in mdlInitializeSizes

The `ssGetPath`, `ssGetModelName`, and `ssSetStatus` macros work in `mdlInitializeSizes`.

mdlDerivatives

Purpose

Compute derivatives for continuous states.

Declaration

```
static void mdlDerivatives(
   double *dx, double *x, double *u, SimStruct *S, int tid)
```

Arguments

`double *dx`	The derivatives vector (returned)
`double *x`	The states vector (both continuous and discrete)
`double *u`	The input vector
`SimStruct *S`	The `SimStruct` for this block
`int tid`	The task ID (for use with multitasking real-time operating systems)

Description

The `mdlDerivatives` function computes the derivatives for the S-function's continuous states, where the derivative is a function of time, the states, and the input, as shown in the equation below.

$$\dot{x}_c = f_d(t,x,u)$$

Simulink calls the `mdlDerivatives` function at each major and minor integration step. The input arguments are the states at the previous time value, the current inputs, the `SimStruct`, and a task ID. The task ID is used when the system is executing in a multithreaded environment.

Example

```
static void mdlDerivatives(
   double *dx, double *x, double *u, SimStruct *S, int tid)
{
   int i;

   /*
    * Compute the derivatives for a basic integrator
    * where dx is equal to the input
    */

   for (i = 0; i < ssGetNumContStates(S); i++)
      dx[i] = u[i];
}
```

mdlInitializeConditions

Purpose Initialize states and work vectors.

Declaration `static void mdlInitializeConditions(double *x0, SimStruct *S)`

Arguments

`double *x0`	The initial states (continuous and discrete) vector (returned)
`SimStruct *S`	The `SimStruct` for this block

Description The `mdlInitializeConditions` function initializes the S-function's states and work vectors. Simulink calls this function at the beginning of the simulation. The initial states are defined in parameter `x0`. For hybrid blocks, continuous states precede discrete states, as described in the illustration in "Setting the Initial Conditions" on page 8–47.

The work vectors (if any) have already been allocated by Simulink and initialized to all zeros. Any additional initialization should be done in this function.

For more information about setting initial conditions, see page 8-47.

Example

```
static void mdlInitializeConditions(
   double *x0, SimStruct *S)
{
   /*
    * Get real (RWork) and integer (IWork) work vectors
    */

   double *RWork;
   int *IWork, i, dsIndex, nStates;

   /*
    * Initialize the continuous states to all ones.
    */

   for (i = 0; i < ssGetNumContStates(S); i++)
      x0[i] = 1.0;
```

```
    /*
     * Initialize all the discrete states to minus one.
     * Note, the discrete states immediately follow
     * the continuous states in the x0 vector.
     */

    dsIndex = ssGetNumContStates(S);
    nStates = ssGetNumTotalStates(S);
    for (i=dsIndex; i<nStates; i++)
       x0[i] = –1.0;

    /*
     * Initialize the real work vector to 2.0
     */

    RWork = ssGetRWork(S);
    for (i = 0; i < ssGetNumRWork(S); i++)
       RWork[i] = 2.0;

    /*
     * Initialize the integer work vector to 3
     */

    IWork = ssGetIWork(S);
    for (i = 0; i < ssGetNumIWork(S); i++)
       IWork[i] = 3;
}
```

mdlInitializeSampleTimes

Purpose Initialize sample times array.

Declaration

```
static void mdlInitializeSampleTimes(SimStruct *S)
```

Arguments

`SimStruct *S`	The `SimStruct` for this block

Description

The `mdlInitializeSampleTimes` function initializes the sample times and offsets arrays stored in the `SimStruct`. Simulink uses these arrays to compute when the next sample hit will occur so the solver stops at that time. The sample times and offsets are set using the following macros:

```
ssSetSampleTimeEvent(S, st_index, sample_time)
ssSetOffsetTimeEvent(S, st_index, offset_time)
```

where the arguments are as follows:

`S`	The `SimStruct`.
`st_index`	When setting a sample time, `st_index` is the array element of the sample time. To set one sample time, specify an `st_index` value of 0; to set two sample times, invoke the macro twice, with `st_index` values of 0 and 1, and so on. When setting an offset, `st_index` corresponds to the index of the sample time to which the offset applies.
`sample_time`	The sample time for the `st_index` element.
`offset_time`	The offset for the `st_index` element.

If the S-function is purely continuous in nature, you should set the first sample time and offset to zero.

The number of sample times set in this function must match the number of sample times set by the `ssSetNumSampleTimes` macro in the `mdlInitializeSizes` function. For more information about setting sample times and offsets, see page 8-45.

Example

```
static void mdlInitializeSampleTimes(SimStruct *S)
{
   /*
    * Initialize continuous sample time to 0.0
    * Initialize offset to 0.0
    */
   ssSetSampleTimeEvent(S, 0, 0.0);
   ssSetOffsetTimeEvent(S, 0, 0.0);

   /*
    * Initialize discrete sample time to 0.1 secs
    * Initialize offset for discrete ST to 0.025 secs
    */
   ssSetSampleTimeEvent(S, 1, 0.1);
   ssSetOffsetTimeEvent(S, 1, 0.025);
}
```

mdlInitializeSizes

Purpose Initialize sizes structure.

Declaration `static void mdlInitializeSizes(SimStruct *S)`

Arguments `SimStruct *S` The SimStruct for this block

Description The mdlInitializeSizes function initializes the sizes structure stored in the SimStruct. You define values for the sizes structure fields using the SimStruct access macros. All undefined sizes fields are set to zero.

To specify that certain block characteristics be dependent on the number of inputs to the S-Function block, set the value of the sizes field to –1 or use the DYNAMICALLY_SIZED macro.

For more information about using this function, see “Defining S-Function Block Characteristics” on page 8–28.

Example

```
static void mdlInitializeSizes(SimStruct *S)
{
   ssSetNumContStates         (S, 2);   /* continuous states */
   ssSetNumDiscStates         (S, 1);   /* discrete states */
   ssSetNumInputs             (S, 2);   /* inputs */
   ssSetNumOutputs            (S, 2);   /* outputs */
   ssSetDirectFeedThrough     (S, 0);   /* direct feedthrough*/
   ssSetNumRWork              (S, 3);   /* real work vector */
   ssSetNumIWork              (S, 3);   /* integer work vect */
   ssSetNumPWork              (S, 0);   /* pointer work vect */
   ssSetNumInputArgs          (S, 1);   /* input arguments */
   ssSetNumSampleTimes        (S, 3);   /* sample times */
}
```

Purpose Compute output vector.

Declaration

```
static void mdlOutputs(
double *y, double *x, double *u, SimStruct *S, int tid)
```

Arguments

`double *y`	The output vector (returned)
`double *x`	The states vector
`double *u`	The input vector
`SimStruct *S`	The `SimStruct` for this block
`int tid`	The task ID (for use with multitasking real-time operating systems)

Description The `mdlOutputs` function computes the S-Function block output, which is a function of time, the states, and the input, as shown in the equation below.

$$y = f_O(t, x, u)$$

The input arguments are the current states, the current inputs, the `SimStruct`, and a task ID. The task ID is used when the system is executing in a multithreaded environment.

Example

```
static void mdlOutputs(
   double *y, double *x, double *u, SimStruct *S, int tid)
{
   int i;

   /*
    * Compute the outputs that are a function of
    * the continuous states
    */

   for (i = 0; i < ssGetNumContStates(S); i++)
      y[i] = x[i];
```

```
      /*
       * Compute the outputs that are a function of
       * the discrete states
       */

      if (ssIsSampleHitEvent(S, 0, tid)) {
         int dsIndex, nStates;
         dsIndex = ssGetNumContStates(S);
         nStates = ssGetNumTotalStates(S);

         for (i=dsIndex; i<nStates; i++)
            y[i] = x[i];
      }
   }
```

Purpose Clean up at termination of the simulation.

Declaration

```
static void mdlTerminate(SimStruct *S)
```

Arguments

`SimStruct *S`	The `SimStruct` for this block

Description The `mdlTerminate` function performs any necessary actions upon termination of the simulation, such as deallocating memory that was allocated in the `mdlInitializeConditions` function.

Example For an example of the use of the `mdlTerminate` function, see “Allocating Work Vectors and Setting Their Values” on page 8–48.

mdlUpdate

Purpose

Major time step update.

Declaration

```
static void mdlUpdate(
double *x, double *u, SimStruct *S, int tid)
```

Arguments

`double *x`	The states vector, both continuous and discrete (input and returned)
`double *u`	The input vector
`SimStruct *S`	The `SimStruct` for this block
`int tid`	The task ID (for use with multitasking real-time operating systems

Description

The `mdlUpdate` function is called once at each major integration step, such as for updating the discrete state vector.

$$x_{d_{k+1}} = f_u(t, x, u)$$

The function is also useful for S-functions that need to perform some action at each major integration step, such as data collection blocks. Note that you can also change the values of continuous states in this function.

An S-function's `mdlUpdate` function will be called if and only if the S-function has one or more discrete states or does *not* have direct feedthrough. If your S-function needs to have its `mdlUpdate` routine called and it does not satisfy either of the above conditions, you must specify that it has a discrete state using the `ssSetNumDiscStates` macro in the `mdlInitializeSizes` function.

The inputs to this function are the current states, the current inputs, the `SimStruct`, and a task ID. The task ID is used when the system is executing in a multithreaded environment.

Example

```
static void mdlUpdate(
   double *x, double *u, SimStruct *S, int tid)
{
   /*
    * Compute the discrete states, as just the sampled
    * continuous states. The use of ssIsSampleHitEvent
    * ensures that the discrete states should only be
    * updated on discrete time hits.
    */

    if (ssIsSampleHitEvent(S, 0, tid)) {
      int i;

      for (i = 0; i < ssGetNumDiscStates(S); i++)
         x[ssGetNumContStates(S) + i] = x[i];
   }
}
```

9

Block Reference

What Each Block Reference Page Contains

Blocks appear in alphabetical order and contain this information:

- The block name, icon, and block library that contains the block.
- The purpose of the block.
- A description of the block's use.
- The block dialog box and parameters.
- The block characteristics, including some or all of these, as they apply to the block:
  - Direct Feedthrough – whether the block or any of its ports has direct feedthrough. For more information, see "Algebraic Loops" on page 10–7.
  - Sample Time – how the block's sample time is determined, whether by the block itself (as is the case with discrete and continuous blocks) or inherited from the block that drives it or is driven by it. For more information, see "Sample Time" on page 10–11.
  - Scalar Expansion – whether or not scalar values are expanded to vectors. Some blocks expand scalar inputs and/or parameters as appropriate. For more information, see "Scalar Expansion of Inputs and Parameters" on page 3–11.
  - States – the number of discrete and continuous states.
  - Vectorized – whether the block accepts and/or generates vector signals. For more information, see "Vector Input and Output" on page 3–11.
  - Zero Crossings – whether the block detects state events. For more information, see "Zero Crossings" on page 10–3.

The Simulink Block Libraries

Simulink organizes its blocks into block libraries according to their behavior. The **simulink** window displays the block library icons and names:

- The *Sources* library contains blocks that generate signals.
- The *Sinks* library contains blocks that display or write block output.
- The *Discrete* library contains blocks that describe discrete-time components.
- The *Linear* library contains blocks that describe linear functions.
- The *Nonlinear* library contains blocks that describe nonlinear functions.
- The *Connections* library contains blocks that allow multiplexing and demultiplexing, implement external Input/Output, pass data to other parts of the model, create subsystems, and perform other functions.
- The *Blocksets and Toolboxes* library contains the Extras block library of specialized blocks.
- The *Demos* library contains useful MATLAB and Simulink demos.

Table 9-1: Sources Library Blocks

Block Name	Purpose
Band-Limited White Noise	Introduce white noise into a continuous system.
Chirp Signal	Generate a sine wave with increasing frequency.
Clock	Display and provide the simulation time.
Constant	Generate a constant value.
Digital Clock	Generate simulation time at the specified sampling interval.
Digital Pulse Generator	Generate pulses at regular intervals.
From File	Read data from a file.
From Workspace	Read data from a matrix defined in the workspace.
Pulse Generator	Generate pulses at regular intervals.
Ramp	Generate a constantly increasing or decreasing signal.

Table 9-1: Sources Library Blocks (Continued)

Block Name	Purpose
Random Number	Generate normally distributed random numbers.
Repeating Sequence	Generate a repeatable arbitrary signal.
Signal Generator	Generate various waveforms.
Sine Wave	Generate a sine wave.
Step	Generate a step function.
Uniform Random Number	Generate uniformly distributed random numbers.

Table 9-2: Sinks Library Blocks

Block Name	Purpose
Display	Show the value of the input.
Scope	Display signals generated during a simulation.
Stop Simulation	Stop the simulation when the input is nonzero.
To File	Write data to a file.
To Workspace	Write data to a matrix in the workspace.
XY Graph	Display an X-Y plot of signals using a MATLAB figure window.

Table 9-3: Discrete Library Blocks

Block Name	Purpose
Discrete Filter	Implement IIR and FIR filters.
Discrete State-Space	Implement a discrete state-space system.
Discrete-Time Integrator	Perform discrete-time integration of a signal.
Discrete Transfer Fcn	Implement a discrete transfer function.

Table 9-3: Discrete Library Blocks (Continued)

Block Name	Purpose
Discrete Zero-Pole	Implement a discrete transfer function specified in terms of poles and zeros.
First-Order Hold	Implement a first-order sample-and-hold.
Unit Delay	Delay a signal one sample period.
Zero-Order Hold	Implement zero-order hold of one sample period.

Table 9-4: Linear Library Blocks

Block Name	Purpose
Derivative	Output the time derivative of the input.
Dot Product	Generate the dot product.
Gain	Multiply block input.
Integrator	Integrate a signal.
Matrix Gain	Multiply the input by a matrix.
Slider Gain	Vary a scalar gain using a slider.
State-Space	Implement a linear state-space system.
Sum	Generate the sum of inputs.
Transfer Fcn	Implement a linear transfer function.
Zero-Pole	Implement a transfer function specified in terms of poles and zeros.

Table 9-5: Nonlinear Library Blocks

Block Name	Purpose
Abs	Output the absolute value of the input.
Algebraic Constraint	Constrain the input signal to zero.
Backlash	Model the behavior of a system with play.
Combinatorial Logic	Implement a truth table.
Coulomb & Viscous Friction	Model discontinuity at zero, with linear gain elsewhere.
Dead Zone	Provide a region of zero output.
Fcn	Apply a specified expression to the input.
Hit Crossing	Detect crossing point.
Logical Operator	Perform the specified logical operation on the input.
Look-Up Table	Perform piecewise linear mapping of the input.
Look-Up Table (2-D)	Perform piecewise linear mapping of two inputs.
Manual Switch	Switch between two inputs.
Math Function	Perform a mathematical function.
MATLAB Fcn	Apply a MATLAB function or expression to the input.
Memory	Output the block input from the previous time step.
MinMax	Output the minimum or maximum input value.
Multiport Switch	Choose between block inputs.
Product	Generate the product or quotient of block inputs.
Quantizer	Discretize input at a specified interval.
Rate Limiter	Limit the rate of change of a signal.
Relational Operator	Perform the specified relational operation on the input.
Relay	Switch output between two constants.

Table 9-5: Nonlinear Library Blocks (Continued)

Block Name	Purpose
Rounding Function	Perform a rounding function.
Saturation	Limit the range of a signal.
S-Function	Access an S-function.
Sign	Indicate the sign of the input.
Switch	Switch between two inputs.
Transport Delay	Delay the input by a given amount of time.
Trigonometric Function	Perform a trigonometric function.
Variable Transport Delay	Delay the input by a variable amount of time.

Table 9-6: Connections Library Blocks

Block Name	Purpose
Data Store Memory	Define a shared data store.
Data Store Read	Read data from a shared data store.
Data Store Write	Write data to a shared data store.
Demux	Separate a vector signal into output signals.
Enable	Add an enabling port to a subsystem.
From	Accept input from a Goto block.
Goto	Pass block input to From blocks.
Goto Tag Visibility	Define the scope of a Goto block tag
Ground	Ground an unconnected input port.
IC	Set the initial value of a signal.
Inport	Create an input port for a subsystem or an external input.

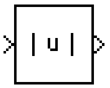
Table 9-6: Connections Library Blocks (Continued)

Block Name	Purpose
Mux	Combine several input lines into a vector line.
Outport	Create an output port for a subsystem or an external output.
Selector	Select input elements.
Subsystem	Represent a system within another system.
Terminator	Terminate an unconnected output port.
Trigger	Add a trigger port to a subsystem.
Width	Output the width of the input vector.

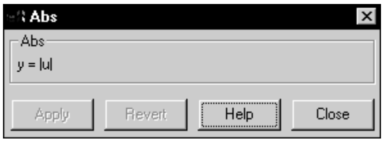
Purpose Output the absolute value of the input.

Library Nonlinear

Description The Abs block generates as output the absolute value of the input. The block accepts one input and generates one output.



Dialog Box



Characteristics

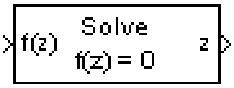
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Vectorized	Yes
Zero Crossing	Yes, to detect zero

Algebraic Constraint

Purpose Constrain the input signal to zero.

Library Nonlinear

Description The Algebraic Constraint block constrains the input signal f(*z*) to zero and outputs an algebraic state *z*. The block outputs the value necessary to produce a zero at the input. The output must affect the input through some feedback path. This enables you to specify algebraic equations for index 1 differential/algebraic systems (DAEs).

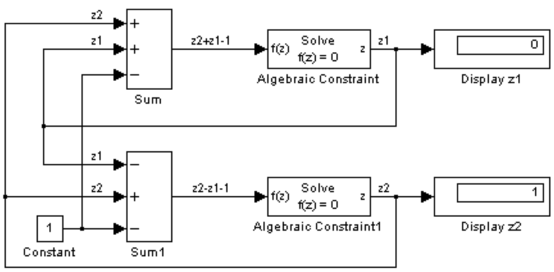


By default, the **Initial guess** parameter is zero. You can improve the efficiency of the algebraic loop solver by providing an **Initial guess** of the algebraic state *z* that is close to the solution value.

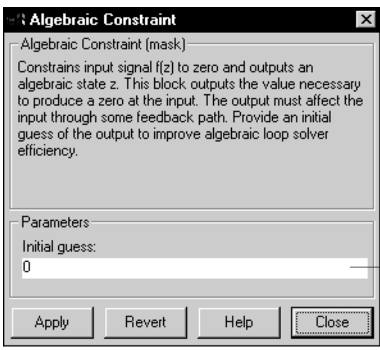
For example, the model below solves these equations:

```
z2 + z1 = 1
z2 – z1 = 1
```

The solution is z2 = 1, z1 = 0, as the Display blocks show.



Parameters and Dialog Box



An initial guess of the solution value. The default is 0.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Vectorized	Yes
Zero Crossing	No

Backlash

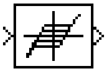
Purpose

Model the behavior of a system with play.

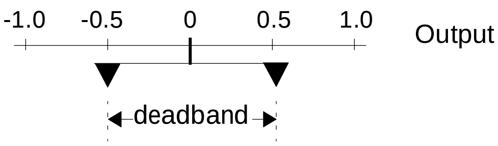
Library

Nonlinear

Description



The Backlash block implements a system in which a change in input causes an equal change in output. However, when the input changes direction, an initial change in input has no effect on the output. The amount of side-to-side play in the system is referred to as the *deadband*. The deadband is centered about the output. This figure shows the block's initial state, with the default deadband width of 1 and initial output of 0:



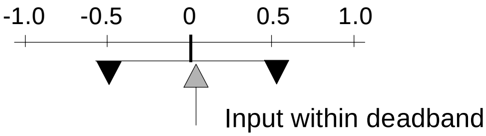
A system with play can be in one of three modes:

- Disengaged – in this mode, the input does not drive the output and the output remains constant.
- Engaged in a positive direction – in this mode, the input is increasing (has a positive slope) and the output is equal to the input *minus* half the deadband width.
- Engaged in a negative direction – in this mode, the input is decreasing (has a negative slope) and the output is equal to the input *plus* half the deadband width.

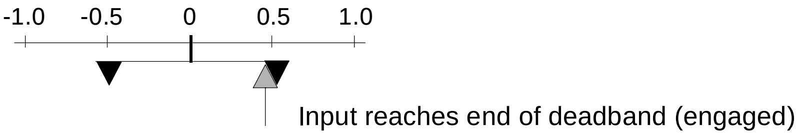
If the initial input is outside the deadband, the **Initial output** parameter value determines if the block is engaged in a positive or negative direction and the output at the start of the simulation is the input plus or minus half the deadband width.

For example, the Backlash block can be used to model the meshing of two gears. The input and output are both shafts with a gear on one end, and the output shaft is driven by the input shaft. Extra space between the gear teeth introduce *play*. The width of this spacing is the **Deadband width** parameter. If the system is disengaged initially, the output (the position of the driven gear) is defined by the **Initial output** parameter.

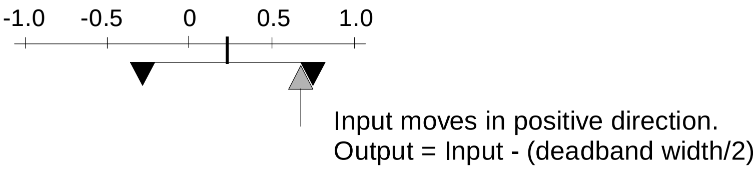
The figures below illustrate the block's operation when the initial input is within the deadband. The first figure shows the relationship between the input and the output while the system is in disengaged mode (and the default parameter values are not changed):



The next figure shows the state of the block when the input has reached the end of the deadband and engaged the output. The output remains at its previous value.



The final figure shows how a change in input affects the output while they are engaged.

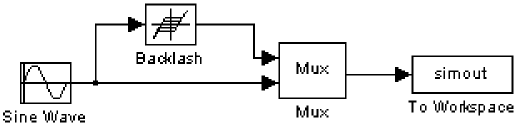


If the input reverses its direction, it disengages from the output. The output remains constant until the input either reaches the opposite end of the deadband or reverses its direction again and engages at the same end of the deadband. Now, as before, movement in the input causes equal movement in the output.

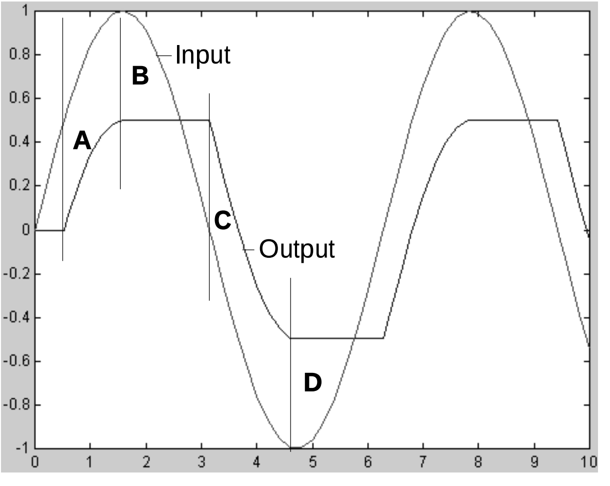
For example, if the deadband width is 2 and the initial output is 5, the output, y, at the start of the simulation is:

- 5 if the input, u, is between 4 and 6
- $u + 1$ if $u < 4$
- $u - 1$ if $u > 6$

This sample model and the plot that follows it show the effect of a sine wave passing through a Backlash block.



The Backlash block parameters are unchanged from their default values (the deadband width is 1 and the initial output is 0). Notice in the plotted output below that the Backlash block output is zero until the input reaches the end of the deadband (at 0.5). Now, the input and output are engaged and the output moves as the input does until the input changes direction (at 1.0). When the input reaches 0, it again engages the output at the opposite end of the deadband.



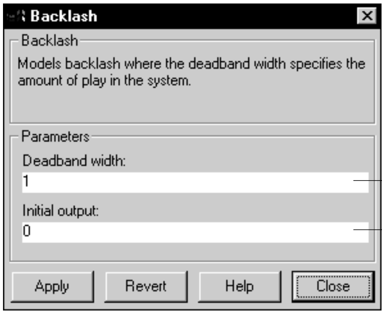
A Input engages in positive direction. Change in input causes equal change in output.

B Input disengages. Change in input does not affect output.

C Input engages in negative direction. Change in input causes equal change in output.

D Input disengages. Change in input does not affect output.

Parameters and Dialog Box



The width of the deadband. The default is 1.

The initial output value. The default is 0.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Vectorized	Yes
Zero Crossing	Yes, to detect engagement with lower and upper thresholds

Band-Limited White Noise

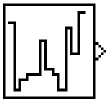
Purpose

Introduce white noise into a continuous system.

Library

Sources

Description



The Band-Limited White Noise block generates normally distributed random numbers that are suitable for use in continuous or hybrid systems.

The primary difference between this block and the Random Number block is that the Band-Limited White Noise block produces output at a specific sample rate, which is related to the correlation time of the noise.

Theoretically, continuous white noise has a correlation time of 0, a flat Power Spectral Density (PSD), and a covariance of infinity. In practice, physical systems are never disturbed by white noise, although white noise is a useful theoretical approximation when the noise disturbance has a correlation time that is very small relative to the natural bandwidth of the system.

In Simulink, you can simulate the effect of white noise by using a random sequence with a correlation time much smaller than the shortest time constant of the system. The Band-Limited White Noise block produces such a sequence. The correlation time of the noise is the sample rate of the block. For accurate simulations, use a correlation time much smaller than the fastest dynamics of the system. You can get good results by specifying:

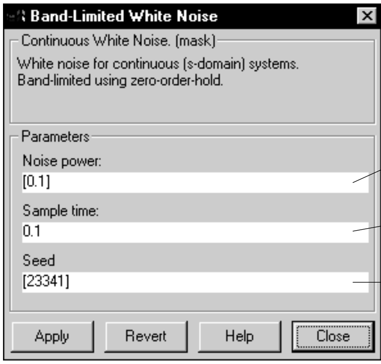
$$t_c \approx \frac{1}{100}\frac{2\pi}{f_{max}}$$

where f_{max} is the bandwidth of the system in rad/sec.

The Algorithm Used in the Block Implementation

To produce the correct intensity of this noise, the covariance of the noise is scaled to reflect the implicit conversion from a continuous PSD to a discrete noise covariance. The appropriate scale factor is $1/t_c$, where t_c is the correlation time of the noise. This scaling ensures that the response of a continuous system to our approximate white noise has the same covariance as the system would have if we had used true white noise. Because of this scaling, the covariance of the signal from the Band-Limited White Noise block is not the same as the **Noise power** (intensity) dialog box parameter. This parameter is actually the height of the PSD of the white noise. While the covariance of true white noise is infinite, the approximation used in this block has the property that the covariance of the block output is the **Noise Power** divided by t_c.

Parameters and Dialog Box



The height of the PSD of the white noise. The default value is 0.1.

The correlation time of the noise. The default value is 0.1.

The starting seed for the random number generator. The default value is 23341.

Characteristics

Sample Time	Discrete
Scalar Expansion	Of **Noise power** and **Seed** parameters and output
Vectorized	Yes
Zero Crossing	No

Chirp Signal

Purpose

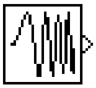
Generate a sine wave with increasing frequency.

Library

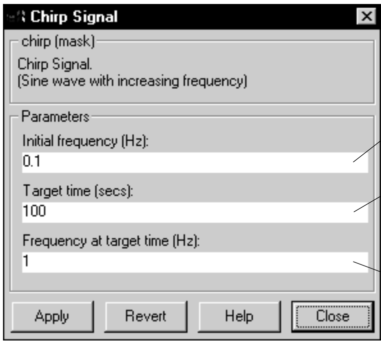
Sources

Description

The Chirp Signal block generates a sine wave whose frequency increases at a linear rate with time. You can use this block for spectral analysis of nonlinear systems. The block generates a scalar or vector output.



Parameters and Dialog Box



The initial frequency of the signal, specified as a scalar or vector value. The default is 0.1 Hz.

The time at which the frequency reaches the **Frequency at target time** parameter value, a scalar or vector value. The frequency continues to change at the same rate after this time. The default is 100 seconds.

The frequency of the signal at the target time, a scalar or vector value. The default is 1 Hz.

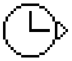
Characteristics

Sample Time	Continuous
Scalar Expansion	Of parameters
Vectorized	Yes
Zero Crossing	No

Purpose Display and provide the simulation time.

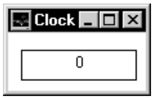
Library Sources

Description The Clock block outputs the current simulation time at each simulation step. When the block is opened, this time is displayed in the window. Running a simulation with this block open slows down the simulation. The Clock block is useful for other blocks that need the simulation time.



When you need the current time within a discrete system, use the Digital Clock block.

Dialog Box



Characteristics

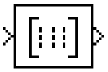
Sample Time	Continuous
Scalar Expansion	N/A
Vectorized	No
Zero Crossing	No

Combinatorial Logic

Purpose Implement a truth table.

Library Nonlinear

Description The Combinatorial Logic block implements a standard truth table for modeling programmable logic arrays (PLAs), logic circuits, decision tables, and other Boolean expressions. You can use this block in conjunction with Memory blocks to implement finite-state machines or flip-flops.



You specify a matrix that defines all possible block outputs as the **Truth table** parameter. Each row of the matrix contains the output for a different combination of input elements. You must specify outputs for every combination of inputs. Each output can be any numerical value; it is not constrained to the values of 0 and 1. The number of columns is the number of block outputs.

The relationship between the number of inputs and the number of rows is:

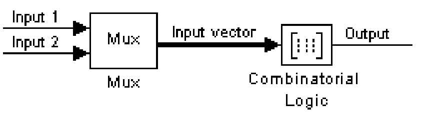
```
number of rows = 2 ^ (number of inputs)
```

Simulink returns a row of the matrix by computing the row's index from the input vector elements. Simulink computes the index by building a binary number where input vector elements having zero values are 0 and elements having nonzero values are 1, then adds 1 to the result. For an input vector, `u`, of `m` elements:

$$\texttt{row index} = 1 + u(m)*2^{0} + u(m-1)*2^{1} + \ldots + u(1)*2^{m}-1$$

Example of Two-Input AND Function

This example builds a two-input AND function, which returns 1 when both input elements are 1, and 0 otherwise. To implement this function, specify the **Truth table** parameter value as `[0; 0; 0; 1]`. The portion of the model that provides the inputs to and the output from the Combinatorial Logic block might look like this:



The table below indicates the combination of inputs that generate each output. The input signal labeled "Input 1" corresponds to the column in the table labeled Input 1. Similarly, the input signal "Input 2" corresponds to the column

with the same name. The combination of these values determines which row of the Output column of the table gets passed as block output.

For example, if the input vector is `[1 0]`, the input references the third row (2^1`*1 + 1`). So, the output value is 0.

Row	Input 1	Input 2	Output
1	0	0	0
2	0	1	0
3	1	0	0
4	1	1	1

Example of Circuit

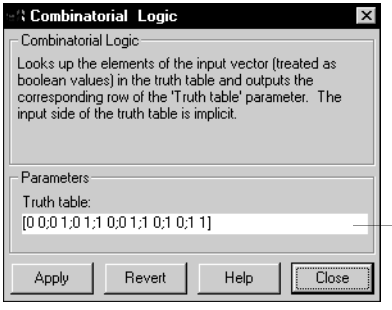
This sample circuit has three inputs: the two bits (**a** and **b**) to be summed and a carry-in bit (**c**). It has two outputs, the carry-out bit (**c'**) and the sum bit (**s**). Here is the truth table and the outputs associated with each combination of input values for this circuit:

Inputs			Outputs	
a	**b**	**c**	**c'**	**s**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

To implement this adder with the Combinatorial Logic block, you enter the 8-by-2 matrix formed by columns **c'** and **s** as the **Truth table** parameter.

Sequential circuits (that is, circuits with states) can also be implemented with the Combinatorial Logic block by including an additional input for the state of the block and feeding the output of the block back into this state input.

Parameters and Dialog Box



The matrix of outputs. Each column corresponds to an element of the output vector and each row corresponds to a row of the truth table.

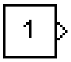
Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Vectorized	Yes; the output width is the number of columns of the **Truth table** parameter
Zero Crossing	No

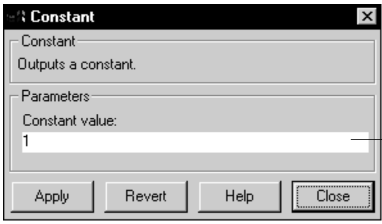
Purpose Generate a constant value.

Library Sources

Description The Constant block generates a specified value independent of time. The block generates one output, which can be scalar or vector, depending on the length of the **Constant value** parameter. The block icon displays the specified value or values.



Parameters and Dialog Box



The output of the block. If a vector, the output is a vector of constants with the specified values. The default value is 1.

Characteristics

Sample Time	Constant
Scalar Expansion	No
Vectorized	Yes
Zero Crossing	No

Coulomb and Viscous Friction

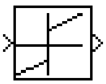
Purpose

Model discontinuity at zero, with linear gain elsewhere.

Library

Nonlinear

Description



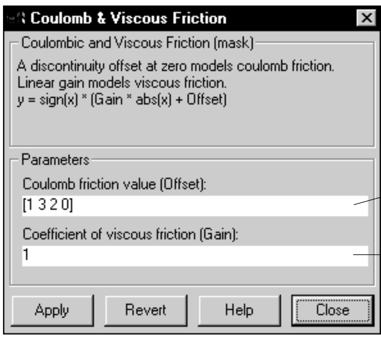
The Coulomb and Viscous Friction block models Coulomb (static) and viscous (dynamic) friction. The block models a discontinuity at zero and a linear gain otherwise. The offset corresponds to the Coulombic friction; the gain corresponds to the viscous friction. The block is implemented as:

```
y = sign(u) * (Gain * abs(u) + Offset)
```

where `y` is the output, `u` is the input, and `Gain` and `Offset` are block parameters.

The block accepts one input and generates one output.

Parameters and Dialog Box



The offset, applied to all input values. The default is `[1 3 2 0]`.

The signal gain at nonzero input points. The default is 1.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Vectorized	Yes
Zero Crossing	Yes, at the point where the static friction is overcome

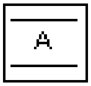
Purpose

Define a data store.

Library

Connections

Description



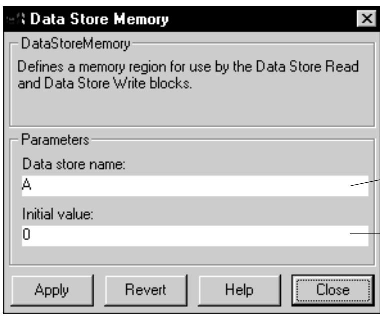
The Data Store Memory block defines and initializes a named shared data store, which is a memory region usable by the Data Store Read and Data Store Write blocks.

Each data store must be defined by a Data Store Memory block. The location of the Data Store Memory block that defines a data store determines the Data Store Read and Data Store Write blocks that can access the data store:

- If the Data Store Memory block is in the *top-level system*, the data store can be accessed by Data Store Read and Data Store Write blocks located anywhere in the model.
- If the Data Store Memory block is in a *subsystem*, the data store can be accessed by Data Store Read and Data Store Write blocks located in the same subsystem or in any subsystem below it in the model hierarchy.

You initialize the data store by specifying values in the **Initial value** parameter. The size of the value determines the size of the data store. An error occurs if a Data Store Write block does not write the same amount of data.

Parameters and Dialog Box



The name of the data store being defined. The default is A.

The initial values of the data store. The default value is 0.

Characteristics

Sample Time	N/A
Vectorized	Yes

Data Store Read

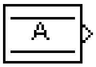
Purpose

Read data from a data store.

Library

Connections

Description

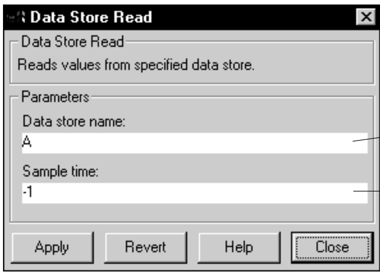


The Data Store Read block reads data from a named data store, passing the data as output. The data was previously initialized by a Data Store Memory block and (possibly) written to that data store by a Data Store Write block.

The data store from which the data is read is determined by the location of the Data Store Memory block that defines the data store. For more information, see page 9–25.

More than one Data Store Read block can read from the same data store.

Parameters and Dialog Box



The name of the data store from which this block reads data.

The sample time, which controls when the block reads from the data store. The default, -1, indicates that the sample time is inherited.

Characteristics

Sample Time	Continuous or discrete
Vectorized	Yes

Purpose

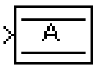
Write data to a data store.

Library

Connections

Description

The Data Store Write block writes the block input to a named data store.

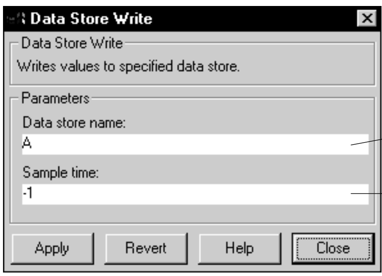


Each write operation performed by a Data Store Write block writes over the data store, replacing the previous contents.

The data store to which this block writes is determined by the location of the Data Store Memory block that defines the data store. For more information, see page 9–25. The size of the data store is set by the Data Store Memory block that defines and initializes the data store. Each Data Store Write block that writes to that data store must write the same amount of data.

More than one Data Store Write block can write to the same data store. However, if two Data Store Write blocks attempt to write to the same data store at the same simulation step, results are unpredictable.

Parameters and Dialog Box



The name of the data store to which this block writes data.

The sample time, which controls when the block writes to the data store. The default, -1, indicates that the sample time is inherited.

Characteristics

Sample Time	Continuous or discrete
Vectorized	Yes

Dead Zone

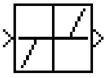
Purpose

Provide a region of zero output.

Library

Nonlinear

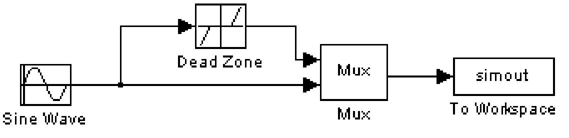
Description



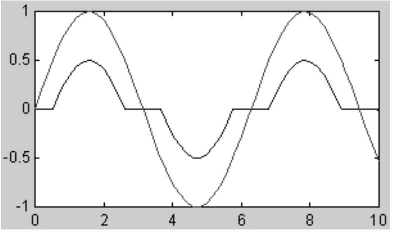
The Dead Zone block generates zero output within a specified region, called its dead zone. The lower and upper limits of the dead zone are specified as the **Start of dead zone** and **End of dead zone** parameters. The block output depends on the input and dead zone:

- If the input is within the dead zone (greater than the lower limit and less than the upper limit), the output is zero.
- If the input is greater than or equal to the upper limit, the output is the input minus the upper limit.
- If the input is less than or equal to the lower limit, the output is the input minus the lower limit.

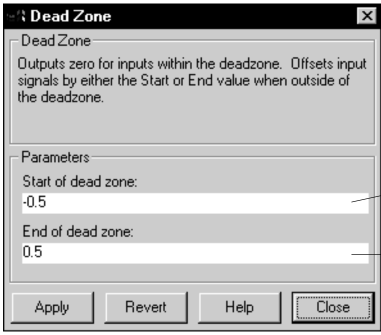
This sample model uses lower and upper limits of -0.5 and +0.5, with a sine wave as input.



This plot shows the effect of the Dead Zone block on the sine wave. While the input (the sine wave) is between -0.5 and 0.5, the output is zero.



Parameters and Dialog Box



The lower limit of the dead zone. The default is -0.5

The upper limit of the dead zone. The default is 0.5.

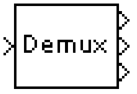
Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of parameters
Vectorized	Yes
Zero Crossing	Yes, to detect when the limits are reached

Purpose Separate a vector signal into output signals.

Library Connections

Description

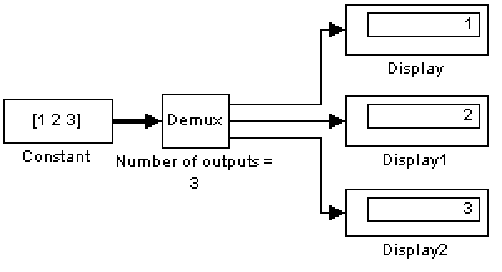


The Demux block separates a vector input signal into output lines, each of which can carry a scalar or vector signal. Simulink determines the number and widths of the output signals by the **Number of outputs** parameter.

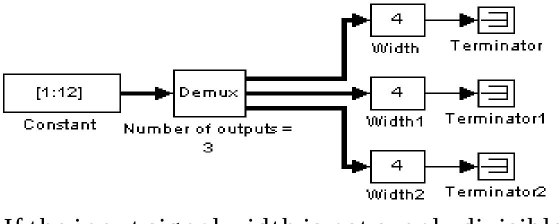
Scalar Number of Outputs

If the **Number of outputs** parameter is a scalar, the block separates the input signal into that number of output signals. The widths of the output signals depend on the width of the input vector and the number of outputs:

- If the input signal width is equal to the number of outputs, the block separates the input signal vector into scalar signals. In this model, the Demux block separates a 3-element vector signal into 3 scalar signals. The **Number of outputs** parameter is 3.

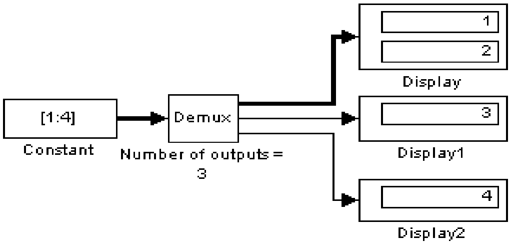


- If the input signal width is evenly divisible by the number of outputs, the block separates the input signal into vector signals of equal width. In this model, the Demux block separates a 12-element vector signal into 3 vector signals, each with a width of 4 elements:



- If the input signal width is not evenly divisible by the number of outputs (and they're not the same), the block separates the input signal into vector signals

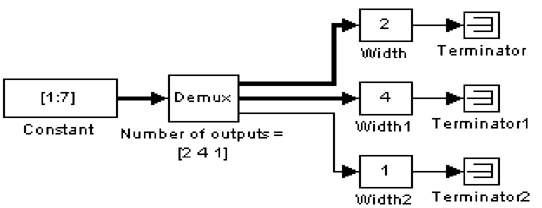
of unequal width and Simulink issues a warning message. In this model, the Demux block separates a 4-element vector signal into 3 signals. The first signal contains the first two elements of the input signal.



Vector Number of Outputs

If the **Number of outputs** parameter is a vector, the number of output lines is equal to the number of elements in the vector. The output signal widths depend on the input vector width and the values of the elements of the parameter. You can explicitly size output signals or let Simulink determine their widths.

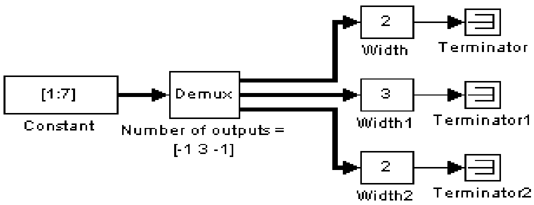
- If the **Number of outputs** vector elements are all positive values, the block generates signals with the specified widths. In this model, the input signal is a vector of width 7 and the **Number of outputs** parameter is `[2 4 1]`:



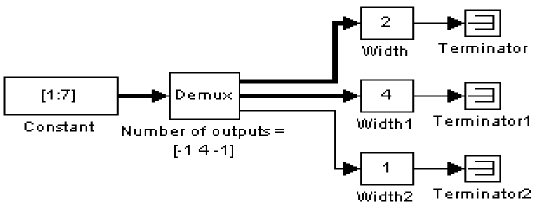
- If the **Number of outputs** vector elements include positive and `–1` values, the block generates output signals with the specified widths for those outputs having positive values and dynamically sizes those outputs having `–1` values.

  In this model, the input signal is a vector of width 7 and the **Number of outputs** parameter is `[–1 3 –1]`. In this example, Simulink explicitly generates a 3-element vector signal as the second output and dynamically

sizes the other two outputs by dividing the remaining input elements as evenly as possible. In this case, the four elements divide equally.



In the next example, the **Number of outputs** is specified as `[-1 4 -1]`. This parameter causes Simulink to generate unequal output vectors:



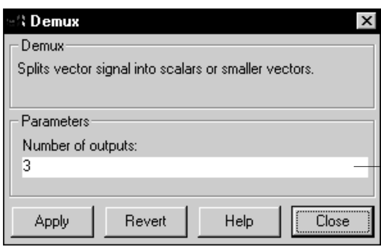
- If the **Number of outputs** vector elements are all `-1`, the number of outputs is equal to the number of vector elements, and the widths are dynamically sized. Specifying the parameter in this way is the same as specifying the parameter as a scalar whose value is the number of elements. For example, entering `[-1 -1 -1]` is the same as specifying a parameter value of `3`.

Simulink draws the specified number of output ports on the block, resizing the block if necessary. When the number of ports is increased or decreased, ports are added or removed from the bottom of the block icon.

Using a Variable to Provide the Number of Outputs Parameter

When you specify the **Number of outputs** parameter as a variable, Simulink issues an error message if the variable is undefined in the workspace.

Parameters and Dialog Box



The number and width of outputs. The total of the output widths must match the width of the input line.

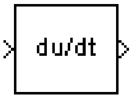
Characteristics	Sample Time	Inherited from driving block
	Vectorized	Yes

Derivative

Purpose Output the time derivative of the input.

Library Linear

Description The Derivative block approximates the derivative of its input by computing



$$\frac{\Delta u}{\Delta t}$$

where Δu is the change in input value and Δt is the change in time since the previous simulation time step. The block accepts one input and generates one output. The value of the input signal before the start of the simulation is assumed to be zero. The initial output for the block is zero.

The accuracy of the results depends on the size of the time steps taken in the simulation. Smaller steps allow a smoother and more accurate output curve from this block. Unlike blocks that have continuous states, the solver does not take smaller steps when the input changes rapidly.

When the input is a discrete signal, the continuous derivative of the input is an impulse when the value of the input changes, otherwise it is 0. You can obtain the discrete derivative of a discrete signal using:

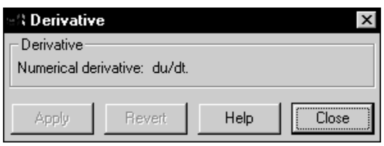
$$y(k) = \frac{1}{\Delta t}(u(k) - u(k-1))$$

Taking the z-transform:

$$\frac{Y(z)}{u(z)} = \frac{1 - z^{-1}}{\Delta t} = \frac{z - 1}{\Delta t \cdot z}$$

Using `linmod` to linearize a model that contains a Derivative block can be troublesome. For information about how to avoid the problem, see "Linearization" on page 5–4.

Dialog Box



Characteristics

Direct Feedthrough	Yes
Sample Time	Continuous
Scalar Expansion	N/A
States	0
Vectorized	Yes
Zero Crossing	No

Digital Clock

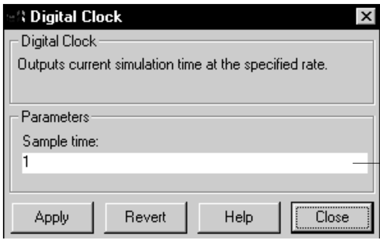
Purpose Output simulation time at the specified sampling interval.

Library Sources

Description The Digital Clock block outputs the simulation time only at the specified sampling interval. At other times, the output is held at the previous value.

Use this block rather than the Clock block (which outputs continuous time) when you need the current time within a discrete system.

Parameters and Dialog Box



The sampling interval. The default value is 1 second.

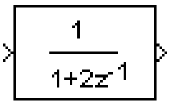
Characteristics

Sample Time	Discrete
Scalar Expansion	No
Vectorized	No
Zero Crossing	No

Purpose Implement IIR and FIR filters.

Library Discrete

Description

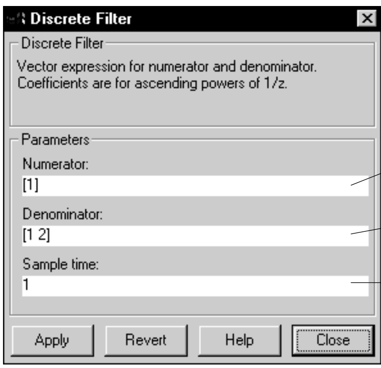


The Discrete Filter block implements Infinite Impulse Response (IIR) and Finite Impulse Response (FIR) filters. You specify the coefficients of the numerator and denominator polynomials in ascending powers of z^{-1} as vectors using the **Numerator** and **Denominator** parameters. The order of the denominator must be greater than or equal to the order of the numerator. See the Discrete Transfer Fcn block description on page 9–48 for more information about coefficients.

The Discrete Filter block represents the method often used by signal processing engineers, who describe digital filters using polynomials in z^{-1} (the delay operator). The Discrete Transfer Fcn block represents the method often used by control engineers, who represent a discrete system as polynomials in z. The methods are identical when the numerator and denominator are the same length. A vector of n elements describes a polynomial of degree n-1.

The block icon displays the numerator and denominator according to how they are specified. For a discussion of how Simulink displays the icon, see the Transfer Fcn block description on page 9–151.

Parameters and Dialog Box



The vector of numerator coefficients. The default is `[1]`.

The vector of denominator coefficients. The default is `[1 2]`.

The time interval between samples.

Characteristics

Direct Feedthrough	Only if the lengths of the **Numerator** and **Denominator** parameters are equal
Sample Time	Discrete
Scalar Expansion	No
States	Length of **Denominator** parameter -1
Vectorized	No
Zero Crossing	No

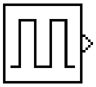
Purpose

Generate pulses at regular intervals.

Library

Sources

Description

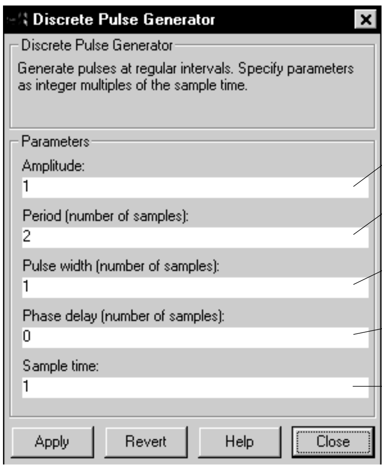


The Discrete Pulse Generator block generates a series of pulses at regular intervals.

The pulse width is the number of sample periods the pulse is high. The period is the number of sample periods the pulse is high and low. The phase delay is the number of sample periods before the pulse starts. The phase delay can be positive or negative but must not be larger than the period. The sample time must be greater than zero.

Use the Discrete Pulse Generator block for discrete or hybrid systems. To generate continuous signals, use the Pulse Generator block, described on page 9–108.

Parameters and Dialog Box



The amplitude of the pulse. The default is 1.

The pulse period in number of samples. The default is 2.

The number of sample periods that the pulse is high. The default is 1.

The delay before each pulse is generated, in number of samples. The default is 0.

The sample period. The default is 1.

Characteristics

Sample Time	Discrete
Scalar Expansion	Of parameters
Vectorized	Yes
Zero Crossing	No

Discrete State-Space

Purpose Implement a discrete state-space system.

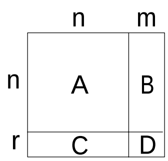
Library Discrete

Description The Discrete State-Space block implements the system described by

$$x(n+1) = Ax(n) + Bu(n)$$
$$y(n) = Cx(n) + Du(n)$$

where u is the input, x is the state, and y is the output. The matrix coefficients must have these characteristics, as illustrated in the diagram below:

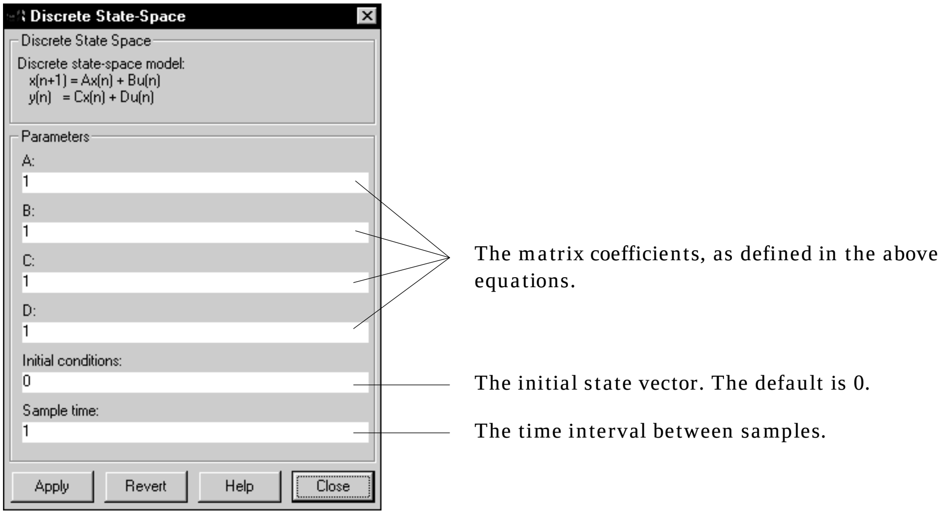
- **A** must be an n-by-n matrix, where n is the number of states.
- **B** must be an n-by-m matrix, where m is the number of inputs.
- **C** must be an r-by-n matrix, where r is the number of outputs.
- **D** must be an r-by-m matrix.



The block accepts one input and generates one output. The input vector width is determined by the number of columns in the B and D matrices. The output vector width is determined by the number of rows in the C and D matrices.

Simulink converts a matrix containing zeros to a sparse matrix for efficient multiplication.

Parameters and Dialog Box



Characteristics

Direct Feedthrough	Only if $D \neq 0$
Sample Time	Discrete
Scalar Expansion	Of the initial conditions
States	Determined by the size of A
Vectorized	Yes
Zero Crossing	No

Purpose

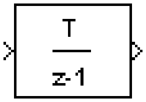
Perform discrete-time integration of a signal.

Library

Discrete

Description

The Discrete-Time Integrator block can be used in place of the Integrator block when constructing a purely discrete system.



The Discrete-Time Integrator block allows you to:

- Define initial conditions on the block dialog box or as input to the block.
- Output the state.
- Define upper and lower limits on the integral.
- Reset the state depending on an additional reset input.

These features are described below.

Integration Methods

The block can integrate using these methods: Forward Euler, Backward Euler, and Trapezoidal. For a given step `k`, Simulink updates `y(k)` and `x(k+1)`. T is the sampling period (delta T in the case of triggered sampling time). Values are clipped according to upper or lower limits. In all cases, `x(0)=IC` (clipped if necessary).

- Forward Euler method (the default), also known as Forward Rectangular, or left-hand approximation. For this method, `1/s` is approximated by `T/(z–1)`. This gives us `y(k) = y(k–1) + T * u(k–1)`

  Let `x(k) = y(k)`, then we have:

  ```
  x(k+1) = x(k) + T * u(k)      (clip if necessary)
    y(k) = x(k)
  ```

  With this method, input port 1 does not have direct feedthrough.

- Backward Euler method, also known as Backward Rectangular or right-hand approximation. For this method, `1/s` is approximated by `T*z/(z–1)`. This gives us `y(k) = y(k–1) + T * u(k)`

  Let `x(k) = y(k–1)`, then we have:

  ```
  x(k+1) = y(k)
    y(k) = x(k) + T * u(k)        (clip if necessary)
  ```

  With this method, input port 1 has direct feedthrough.

- Trapezoidal method. For this method, `1/s` is approximated by `T/2*(z+1)/(z–1)`. This gives us `y(k) = y(k–1) + T/2 * (u(k) + u(k–1))`

  When `T` is fixed (equal to the sampling period), let `x(k) = y(k–1) + T/2 * u(k–1)`, then we have:

  ```
  x(k+1) = y(k) + T/2 * u(k)         (clip if necessary)
    y(k) = x(k) + T/2 * u(k)         (clip if necessary)
  ```

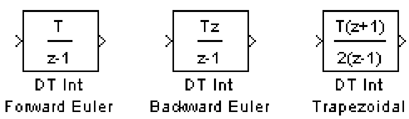
  Here, `x(k+1)` is the best estimate of the next output. It isn't quite the state, in the sense that `x(k) != y(k)`.

  When `T` is variable (that is, obtained from the triggering times), we have:

  ```
  x(k+1) = y(k)
    y(k) = x(k) + T/2 * (u(k) + u(k–1))  (clip if necessary)
  ```

  With this method, input port 1 has direct feedthrough.

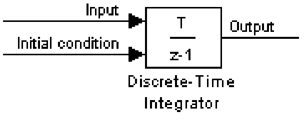
The block icon reflects the selected integration method, as this figure shows:



Defining Initial Conditions

You can define the initial conditions as a parameter on the block dialog box or input them from an external signal:

- To define the initial conditions as a block parameter, specify the **Initial condition source** parameter as **internal** and enter the value in the **Initial condition** parameter field.
- To provide the initial conditions from an external source, specify the **Initial condition source** parameter as **external**. An additional input port appears under the block input, as shown in this figure:



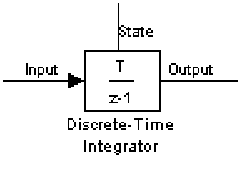
Using the State Port

In two known situations, you must use the state port instead of the output port:

- When the output of the block is fed back into the block through the reset port or the initial condition port, causing an algebraic loop. For an example of this situation, see the bounce model.
- When you want to pass the state from one conditionally executed subsystem to another, which may cause timing problems. For an example of this situation, see the `clutch` model.

You can correct these problems by passing the state through the state port rather than the output port. Although the values are the same, Simulink generates them at slightly different times, which protects your model from these problems. You output the block state by selecting the **Show state port** check box.

By default, the state port appears on the top of the block, as shown in this figure:

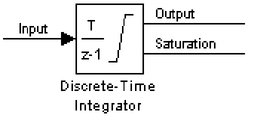


Limiting the Integral

To prevent the output from exceeding specifiable levels, select the **Limit output** check box and enter the limits in the appropriate parameter fields. Doing so causes the block to function as a limited integrator. When the output is outside the limits, the integral action is turned off to prevent integral wind up. During a simulation, you can change the limits but you cannot change whether the output is limited. The output is determined as follows:

- When the integral is less than the **Lower saturation limit** and the input is negative, the output is held at the **Lower saturation limit**.
- When the integral is between the **Lower saturation limit** and the **Upper saturation limit**, the output is the integral.
- When the integral is greater than the **Upper saturation limit** and the input is positive, the output is held at the **Upper saturation limit**.

To generate a signal that indicates when the state is being limited, select the **Show saturation port** check box. A saturation port appears below the block output port, as shown on this figure:



The signal has one of three values:

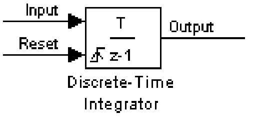
- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- -1 indicates that the lower limit is being applied.

When the **Limit output** option is selected, the block has three zero crossings: one to detect when it enters the upper saturation limit, one to detect when it enters the lower saturation limit, and one to detect when it leaves saturation.

Resetting the State

The block can reset its state to the specified initial condition based on an external signal. To cause the block to reset its state, select one of the **External**

reset choices. A trigger port appears below the block's input port and indicates the trigger type, as shown in this figure.

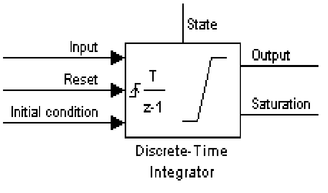


Select **rising** to trigger the state reset when the reset signal has a rising edge. Select **falling** to trigger the state reset when the reset signal has a falling edge. Select **either** to trigger the reset when either a rising or falling signal occurs.

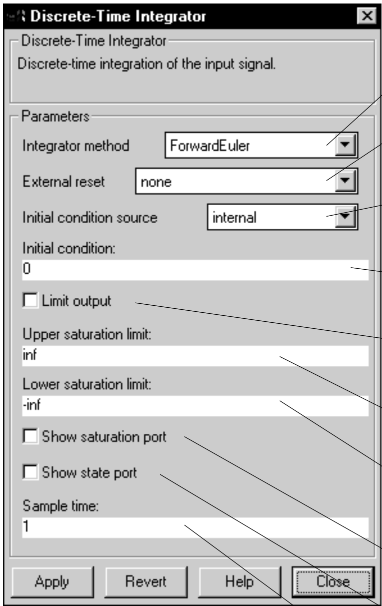
The reset port has direct feedthrough. If the block output is fed back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results. To resolve this loop, feed the block state into the reset port instead. To access the block's state, select the **Show state port** check box.

Choosing All Options

When all options are selected, the icon looks like this:



Parameters and Dialog Box



The integration method. The default is ForwardEuler.

Resets the states to their initial conditions when a trigger event (**rising**, **falling**, or **either**) occurs in the reset signal.

Gets the states' initial conditions from the **Initial condition** parameter (if set to **internal**) or from an external block (if set to **external**).

The states' initial conditions. Set the **Initial condition source** parameter value to **internal**.

If checked, limits the states to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

The upper limit for the integral. The default is `inf`.

The lower limit for the integral. The default is `-inf`.

If checked, adds a saturation output port to the block.

If checked, adds an output port to the block for the block's state.

The time interval between samples. The default is 1.

Characteristics

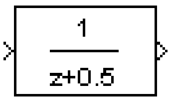
Direct Feedthrough	Yes, of the reset and external initial condition source ports
Sample Time	Discrete
Scalar Expansion	Of parameters
States	Inherited from driving block and parameter
Vectorized	Yes
Zero Crossing	One for detecting reset; one each to detect upper and lower saturation limits, one when leaving saturation

Discrete Transfer Fcn

Purpose Implement a discrete transfer function.

Library Discrete

Description The Discrete Transfer Fcn block implements the z-transform transfer function described by the following equations:



$$H(z) = \frac{num(z)}{den(z)} = \frac{num_0 z^n + num_1 z^{n-1} + \ldots + num_m z^{n-m}}{den_0 z^n + den_1 z^{n-1} + \ldots + den_n}$$

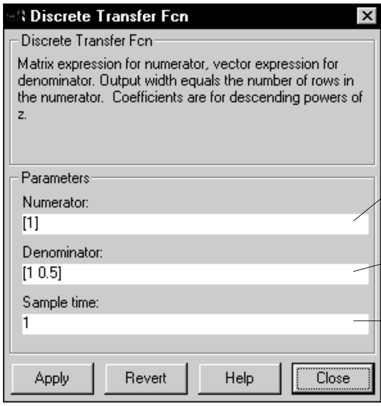
where $m+1$ and $n+1$ are the number of numerator and denominator coefficients, respectively. *num* and *den* contain the coefficients of the numerator and denominator in descending powers of z. *num* can be a vector or matrix, *den* must be a vector, and both are specified as parameters on the block dialog box. The order of the denominator must be greater than or equal to the order of the numerator.

Block input is scalar; output width is equal to the number of rows in the numerator.

The Discrete Transfer Fcn block represents the method typically used by control engineers, representing discrete systems as polynomials in z. The Discrete Filter block represents the method typically used by signal processing engineers, who describe digital filters using polynomials in z^{-1} (the delay operator). The two methods are identical when the numerator is the same length as the denominator.

The Discrete Transfer Fcn block displays the numerator and denominator within its icon depending on how they are specified. See the description of the Transfer Fcn block on page 9–151 for more information.

Parameters and Dialog Box



The row vector of numerator coefficients. A matrix with multiple rows can be specified to generate multiple output. The default is `[1]`.

The row vector of denominator coefficients. The default is `[1 0.5]`.

The time interval between samples. The default is 1.

Characteristics

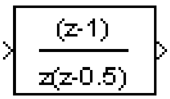
Direct Feedthrough	Only if the lengths of the **Numerator** and **Denominator** parameters are equal
Sample Time	Discrete
Scalar Expansion	No
States	Length of **Denominator** parameter -1
Vectorized	No
Zero Crossing	No

Discrete Zero-Pole

Purpose Implement a discrete transfer function specified in terms of poles and zeros.

Library Discrete

Description The Discrete Zero-Pole block implements a discrete system with the specified zeros, poles, and gain in terms of the delay operator z. A transfer function can be expressed in factored or zero-pole-gain form, which, for a single-input, single-output system in MATLAB, is:

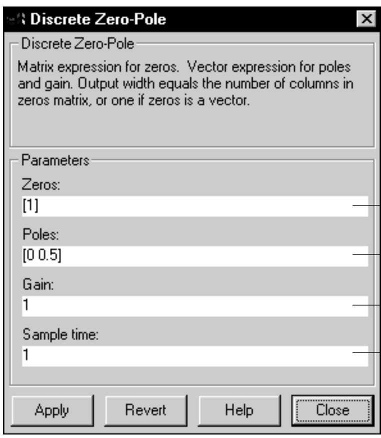


$$H(z) = K\frac{Z(z)}{P(z)} = K\frac{(z - Z_1)(z - Z_2)\ldots(z - Z_m)}{(z - P_1)(z - P_2)\ldots(z - P_n)}$$

where Z represents the zeros vector, P the poles vector, and K the gain. The number of poles must be greater than or equal to the number of zeros ($n \geq m$). If the poles and zeros are complex, they must be complex conjugate pairs.

The block icon displays the transfer function depending on how the parameters are specified. See the description of the Zero-Pole block on page 9–165 for more information.

Parameters and Dialog Box



The matrix of zeros. The default is `[1]`.

The vector of poles. The default is `[0 0.5]`.

The gain. The default is `1`.

The time interval between samples.

Characteristics

Direct Feedthrough	Yes, if the number of zeros and poles are equal
Sample Time	Discrete
Scalar Expansion	No
States	Length of **Poles** vector
Vectorized	No
Zero Crossing	No

Purpose

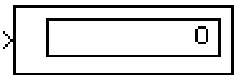
Show the value of the input.

Library

Sinks

Description

The Display block shows the value of its input.



You can control the display format by selecting a **Format** choice:

- `short`, which displays a 5-digit scaled value with fixed decimal point
- `long`, which displays a 15-digit scaled value with fixed decimal point
- `short_e`, which displays a 5-digit value with a floating decimal point
- `long_e`, which displays a 16-digit value with a floating decimal point
- `bank`, which displays a value in fixed dollars and cents format (but with no $ or commas)

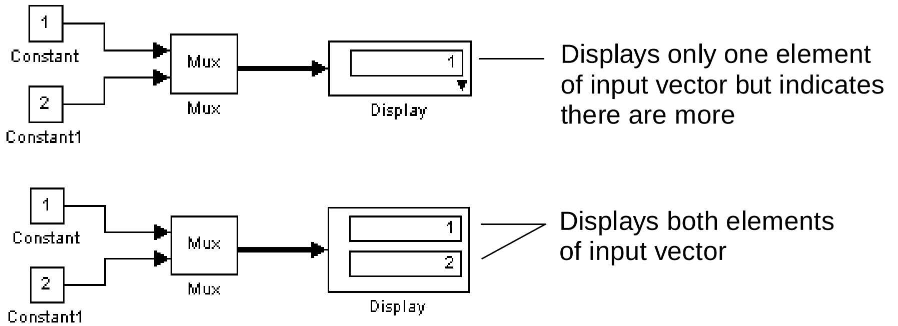
To use the block as a floating display, select the **Floating display** check box. The block's input port disappears and the block displays the value of the signal on a selected line.

The amount of data displayed and the time steps at which the data is displayed are determined by block parameters:

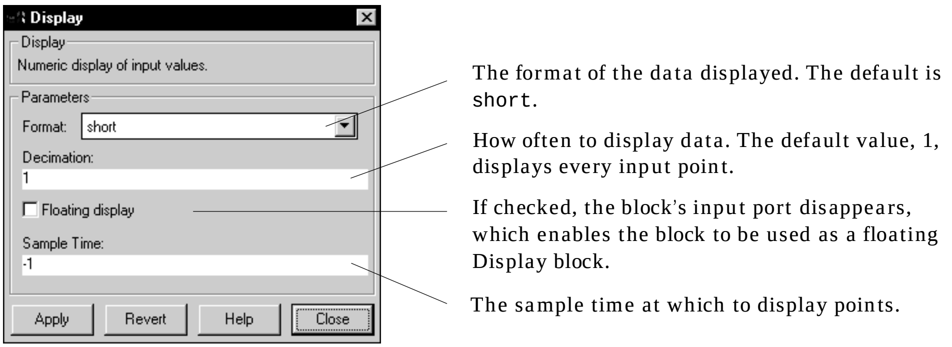
- The **Decimation** parameter enables you to display data at every `n`th sample, where `n` is the decimation factor. The default decimation, 1, displays data at every time step.
- The **Sample time** parameter enables you to specify a sampling interval at which to display points. This parameter is useful when using a variable-step solver where the interval between time steps may not be the same. The default value of `–1` causes the block to ignore sampling interval when determining which points to display.

If the block input is a vector, you can resize the block to show more than just the first element. You can resize the block vertically or horizontally; the block adds display fields in the appropriate direction. A black triangle indicates that the block is not displaying all input vector elements. For example, the figure below shows a model that passes a vector to a Display block. The top model

shows the block before it is resized; notice the black triangle. The bottom model shows the resized block displaying both input elements.



Parameters and Dialog Box



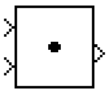
Characteristics

Sample Time	Inherited from driving block
Vectorized	Yes

Purpose Generate the dot product.

Library Linear

Description The Dot Product block generates the dot product of its two input vectors. The scalar output, `y`, is equal to the MATLAB operation

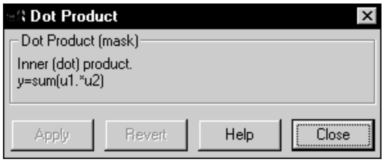


```
y = u1' * u2
```

where `u1` and `u2` represent the vector inputs. If both inputs are vectors, they must be the same length.

To perform element-by-element multiplication without summing, use the Product block.

Dialog Box



Characteristics

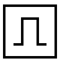
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
States	0
Vectorized	Yes
Zero Crossing	No

Enable

Purpose Add an enabling port to a subsystem.

Library Connections

Description Adding an Enable block to a subsystem makes it an enabled subsystem. An enabled subsystem executes while the input received at the Enable port is greater than zero. For more information about enabled subsystems, see Chapter 7.



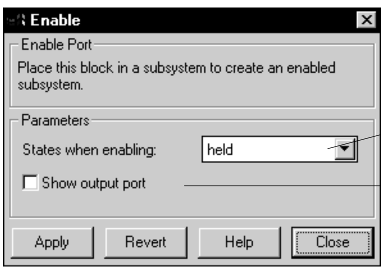
At the start of simulation, Simulink initializes the states of blocks inside an enabled subsystem to their initial conditions. When an enabled subsystem restarts (executes after having been disabled), the **States when enabling** parameter determines what happens to the states of blocks contained in the enabled subsystem:

- **reset** resets the states to their initial conditions (zero if not defined).
- **held** holds the states at their previous values.

You can output the enabling signal by selecting the **Show output port** check box. Selecting this option allows the system to process the enabling signal. The width of the signal is the width of the enabling signal.

A subsystem can contain no more than one Enable block.

Parameters and Dialog Box



Specifies how to handle internal states when the subsystem becomes re-enabled.

If checked, Simulink draws the Enable block output port and outputs the enabling signal.

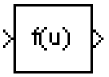
Characteristics

Sample Time	Determined by the signal at the enable port
Vectorized	Yes

Purpose Apply a specified expression to the input.

Library Nonlinear

Description The Fcn block applies the specified C language style expression to its input. The expression can be made up of one or more of these components:



- `u` — the input to the block. If `u` is a vector, `u(i)` represents the `i`th element of the vector; `u(1)` or `u` alone represents the first element.
- Numeric constants
- Arithmetic operators (+ – * /)
- Relational operators (`== != > < >= <=`) — The expression returns 1 if the relation is TRUE; otherwise, it returns 0.
- Logical operators (&& || !) — The expression returns 1 if the relation is TRUE; otherwise, it returns 0.
- Parentheses
- Mathematical functions — `abs`, `acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `exp`, `fabs`, `floor`, `hypot`, `ln`, `log`, `log10`, `pow`, `power`, `rem`, `sign`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh`.
- Workspace variables — Variable names that are not recognized in the list of items above are passed to MATLAB for evaluation. Matrix or vector elements must be specifically referenced (e.g., `A(1,1)` instead of `A` for the first element in the matrix).

The rules of precedence obey the C language standards:

1. ( )
2. `+ -` (unary)
3. `pow` (exponentiation)
4. `!`
5. `* /`
6. `+ -`
7. `> < <= >=`
8. `= !=`

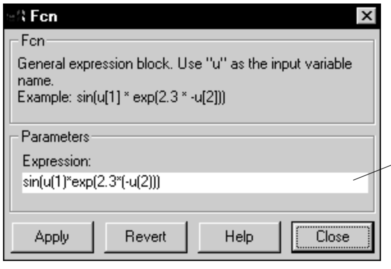
9 &&

10 ||

The expression differs from a MATLAB expression in that the expression cannot perform matrix computations. Also, this block does not support the colon operator (:).

Block input can be a scalar or vector. The output is always a scalar. For vector output, consider using the Math Function block. If a block is a vector and the function operates on input elements individually (for example, the `sin` function), the block operates on only the first vector element.

Parameters and Dialog Box



The C language style expression applied to the input. Expression components are listed above. The expression must be mathematically well formed (i.e., matched parentheses, proper number of function arguments, etc.).

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Vectorized	No
Zero Crossing	No

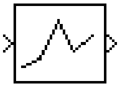
Purpose

Implement a first-order sample-and-hold.

Library

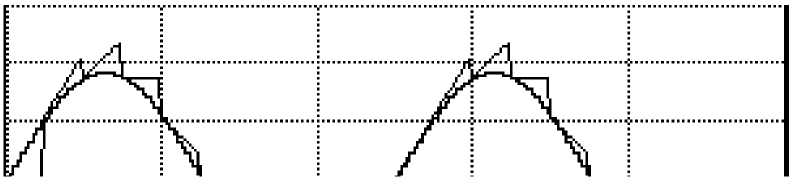
Discrete

Description

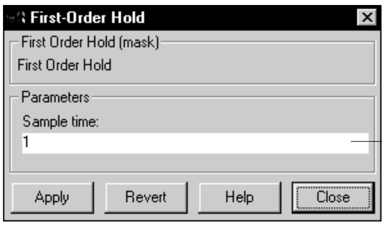


The First-Order Hold block implements a first-order sample-and-hold that operates at the specified sampling interval. This block has little value in practical applications and is included primarily for academic purposes.

You can see the difference between the Zero-Order Hold and First-Order Hold blocks by running the demo program `fohdemo`. This figure compares the output from a Sine Wave block and a First-Order Hold block:



Parameters and Dialog Box



The time interval between samples.

Characteristics

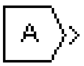
Direct Feedthrough	No
Sample Time	Continuous
Scalar Expansion	No
States	1 continuous and 1 discrete per input element
Vectorized	Yes
Zero Crossing	No

From

Purpose Accept input from a Goto block.

Library Connections

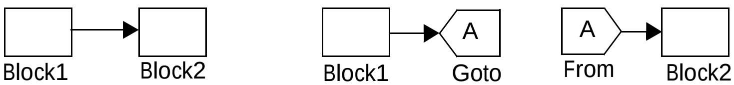
Description



The From block accepts a signal from its corresponding Goto block, then passes it as output. From and Goto blocks allow you to pass a signal from one block to another without actually connecting them. Each From block is associated with a Goto block. The input to that Goto block is passed to the From block, which then passes it to the block connected to it. To associate a Goto block with a From block, enter the Goto block's tag in the **Goto tag** parameter.

A From block can receive its signal from only one Goto block, although a Goto block can pass its signal to more than one From block.

This figure shows that using a Goto block and a From block is equivalent to connecting the blocks to which those blocks are connected. In the model at the left, Block1 passes a signal to Block2. That model is equivalent to the model at the right, which connects Block1 to the Goto block, passes that signal to the From block, then on to Block2.



Associated Goto and From blocks can appear anywhere in a model with this exception: if either block is in a conditionally executed subsystem, the other block must be either in the same subsystem or in a subsystem below it in the model hierarchy (but not in another conditionally executed subsystem). However, if a Goto block is connected to a state port, the signal can be sent to a From block inside another conditionally executed subsystem. For more information about conditionally executed subsystems, see Chapter 7.

The visibility of a Goto block tag determines the From blocks that can receive its signal. For more information, see the descriptions of the Goto block, on page 9–71, and the Goto Tag Visibility block, on page 9–73. The block icon indicates the visibility of the Goto block tag:

- A local tag name is enclosed in square brackets ([]).
- A scoped tag name is enclosed in braces ({}).
- A global tag name appears without additional characters.

Parameters and Dialog Box

The tag of the Goto block passing the signal to this From block.

Characteristics

Sample Time	Inherited from block driving the Goto block
Vectorized	Yes

From File

Purpose

Read data from a file.

Library

Sources

Description

The From File block outputs data read from the specified file. The block icon displays the name of the file supplying the data.

The file must contain a matrix of two or more rows. The first row must contain monotonically increasing time points. Other rows contain data points that correspond to the time point in that column. The matrix is expected to have this form:

$$\begin{bmatrix} t_1 & t_2 & \ldots t_{final} \\ u1_1 & u1_2 & \ldots u1_{final} \\ \ldots & & \\ un_1 & un_2 & \ldots un_{final} \end{bmatrix}$$

The width of the output depends on the number of rows in the file. The block uses the time data to determine its output, but does not output the time values. This means that in a matrix containing m rows, the block outputs a vector of length m–1, consisting of data from all but the first row of the appropriate column.

If an output value is needed at a time that falls between two values in the file, the value is linearly interpolated between the appropriate values. If the required time is less than the first time value or greater than the last time value in the file, Simulink extrapolates using the first two or last two points to compute a value.

If the matrix includes two or more columns at the same time value,the output is the data point for the first column encountered. For example, for a matrix that has this data:

```
time values:    0 1 2 2
data points:    2 3 4 5
```

At time 2, the output is 4, the data point for the first column encountered at that time value.

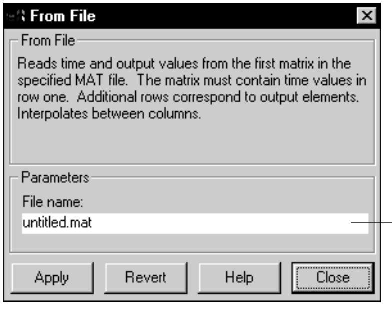
Simulink reads the file into memory at the start of the simulation. As a result, you cannot read data from the same file named in a To File block in the same model.

Using Data Saved by a To File or a To Workspace Block

The From File block can read data written by a To File block without any modifications. To read data written by a To Workspace block and saved to a file:

- The data must include the simulation times. The easiest way to include time data in the simulation output is to specify a variable for time on the **Workspace I/O** page of the **Simulation Parameters** dialog box. See Chapter 4 for more information.
- The form of the data as it is written to the workspace is different from the form expected by the From File block. Before saving the data to a file, transpose it. When it is read by the From File block, it will be in the correct form.

Parameters and Dialog Box



The name of the file that contains the data used as input. The default file name is `untitled.mat`.

Characteristics

Sample Time	Inherited from driven block
Scalar Expansion	No
Vectorized	Yes
Zero Crossing	No

From Workspace

Purpose Read data from a matrix defined in the workspace.

Library Sources

Description The From Workspace block reads data from a matrix in the workspace. The block icon displays the name of the matrix or its time and input components.

The matrix must have two or more columns. The first column must contain monotonically increasing time points. Other columns contain data points that correspond to the time point in that row. The matrix is expected to have this form:

$$\begin{bmatrix} t_1 & u1_1 & \ldots un_1 \\ t_2 & u1_2 & \ldots un_2 \\ \ldots & & \\ t_{final} & u1_{final} & \ldots un_{final} \end{bmatrix}$$

To input data written by a To Workspace block requires that time be added to the matrix. For more information, see the description of the To Workspace block on page 9–148.

The block uses the time data to determine its output, but does not output it. This means that for a matrix containing n columns, the block outputs a vector of length n–1, consisting of data from all but the first column of the appropriate row.

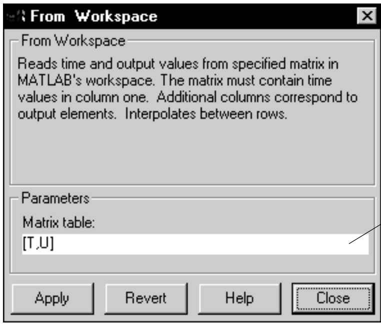
If an output value is needed at a time that falls between two values, the output is linearly interpolated between the two time values that bracket the required time. If the required time is less than the first, or greater than the last time value in the matrix, Simulink extrapolates using the first or last two points.

If the matrix includes two or more rows at the same time value, the output is the data point for the first row encountered. For example, for a matrix that has this data:

```
time values:    0 1 2 2
data points:    2 3 4 5
```

At time 2, the output is 4, the data point for the first column encountered at that time value.

Parameters and Dialog Box



The matrix from which time and data values are read. If these values are not in the same matrix, specify the time column vector `T` and data matrix `U` as `[T,U]`. If the values are in the same matrix, specify the matrix name. Avoid using `ans` to specify the data.

Characteristics

Sample Time	Inherited from driven block
Scalar Expansion	No
Vectorized	Yes
Zero Crossing	No

Gain

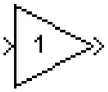
Purpose

Multiply block input.

Library

Linear

Description



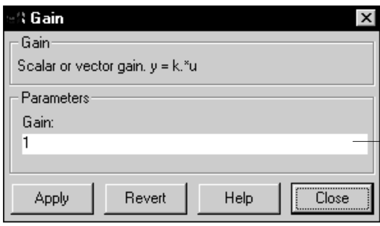
The Gain block generates its output by multiplying its input by a specified constant, variable, or expression.

The gain can be a scalar or vector, which you can enter as a numeric value, or as a variable or expression. To multiply the input by a matrix, use the Matrix Gain block, described on page 9–96.

The Gain block icon displays the value entered in the **Gain** parameter field if the block is large enough. If the gain is specified as a variable, the block displays the variable name, although if the variable is specified in parentheses, the block evaluates the variable each time the block is redrawn and displays its value. If the **Gain** parameter value is too long to be displayed in the block, the string `-K-` is displayed.

To be able to modify the gain during a simulation using a slider control, use the Slider Gain block, described on page 9–136.

Parameters and Dialog Box



The gain, specified as a scalar, vector, variable name, or expression. The default is 1.

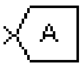
Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of input and **Gain** parameter
States	0
Vectorized	Yes
Zero Crossing	No

Purpose Pass block input to From blocks.

Library Connections

Description



The Goto block passes its input to its corresponding From blocks. From and Goto blocks allow you to pass a signal from one block to another without actually connecting them.

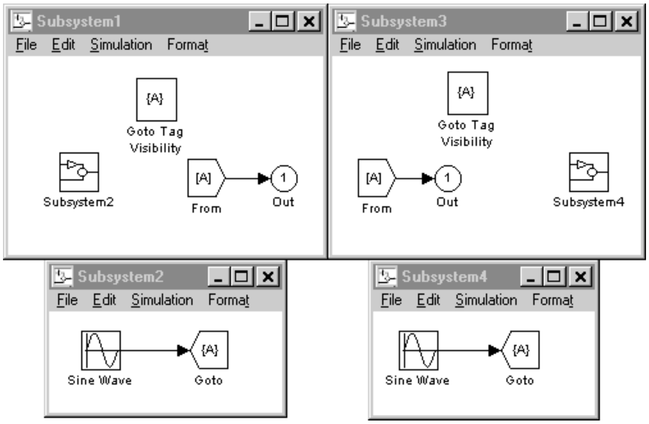
A Goto block can pass its input signal to more than one From block, although a From block can receive a signal from only one Goto block. The input to that Goto block is passed to the From blocks associated with it as though the blocks were physically connected. For limitations on the use of From and Goto blocks, see the description of the From block on page 9–64. Goto blocks and From blocks are matched by the use of Goto tags, defined as the **Tag** parameter.

The **Tag visibility** parameter determines whether the location of From blocks that access the signal is limited:

- **local**, the default, means that From and Goto blocks using the tag must be in the same subsystem. A local tag name is enclosed in square brackets ([]).
- **scoped** means that From and Goto blocks using the same tag must be in the same subsystem or in any subsystem below the Goto Tag Visibility block in the model hierarchy. A scoped tag name is enclosed in braces ({}).
- **global** means that From and Goto blocks using the same tag can be anywhere in the model.

Use local tags when the Goto and From blocks using the same tag name reside in the same subsystem. You must use global or scoped tags when the Goto and From blocks using the same tag name reside in different subsystems. When you define a tag as global, all uses of that tag access the same signal. A tag

defined as scoped can be used in more than one place in the model. This example shows a model that uses two scoped tags with the same name (A):



Parameters and Dialog Box

The Goto block identifier. This parameter identifies the Goto block whose scope is defined in this block.

The scope of the Goto block tag: **local**, **scoped**, or **global**. The default is **local**.

Characteristics

Sample Time	Inherited from driving block
Vectorized	Yes

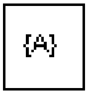
Purpose

Define scope of Goto block tag.

Library

Connections

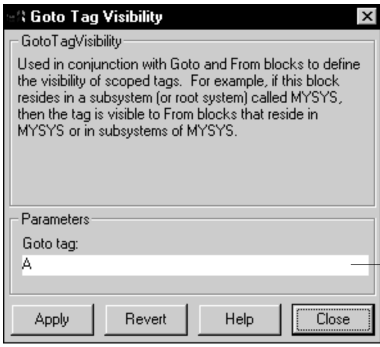
Description



The Goto Tag Visibility block defines the accessibility of Goto block tags that have **scoped** visibility. The tag specified as the **Goto tag** parameter is accessible by From blocks in the same subsystem that contains the Goto Tag Visibility block and in subsystems below it in the model hierarchy.

A Goto Tag Visibility block is required for Goto blocks whose **Tag visibility** parameter value is **scoped**. It is not used if the tag visibility is either **local** or **global**. The block icon shows the tag name enclosed in braces ({}).

Parameters and Dialog Box



The Goto block tag whose visibility is defined by the location of this block.

Characteristics

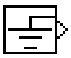
Sample Time	N/A
Vectorized	N/A

Ground

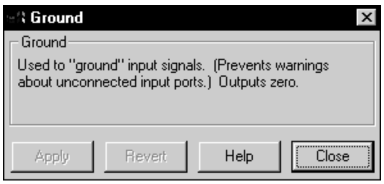
Purpose Ground an unconnected input port.

Library Connections

Description The Ground block can be used to connect blocks whose input ports are not connected to other blocks. If you run a simulation with blocks having unconnected input ports, Simulink issues warning messages. Using Ground blocks to “ground” those blocks avoids warning messages. The Ground block outputs a signal with zero value.



Parameters and Dialog Box



Characteristics

Sample Time	Inherited from driven block
Vectorized	Yes

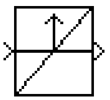
Purpose

Detect crossing point.

Library

Nonlinear

Description



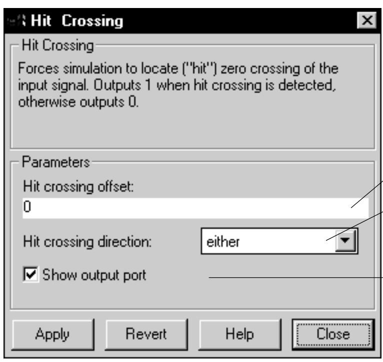
The Hit Crossing block detects when the input reaches the **Hit crossing offset** parameter value in the direction specified by the **Hit crossing direction** parameter. This block locates transitions to, from, and through the offset. The block finds the crossing point to within machine tolerance.

The block has one input. If the **Show output port** check box is selected, the block output indicates when the crossing occurs. If the input signal is exactly the value of the offset value, the block outputs a value of 1 at that time step. If the input signals at two adjacent points bracket the offset value (but neither value is exactly equal to the offset), the block outputs a value of 1 at the second time step. If the **Show output port** check box is *not* selected, the block ensures that the simulation finds the crossing point but does not generate output.

The Hit Crossing block serves as an "Almost Equal" block, useful in working around limitations in finite mathematics and computer precision. Used for these reasons, this block may be more convenient than adding logic to your model to detect this condition.

The `hardstop` and `clutch` demos illustrate the use of the Hit Crossing block. In the `hardstop` demo, the Hit Crossing block is in the Friction Model subsystem. In the `clutch` demo, the Hit Crossing block is in the Lockup Detection subsystem.

Parameters and Dialog Box



The value whose crossing is to be detected.

The direction from which the input signal approaches the hit crossing offset for a crossing to be detected.

Whether to draw an output port.

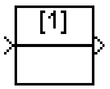
Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Vectorized	Yes
Zero Crossing	Yes, to detect the crossing

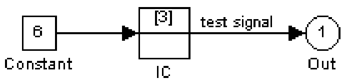
Purpose Set the initial value of a signal.

Library Connections

Description The IC block sets the initial condition of the signal connected to its output port.



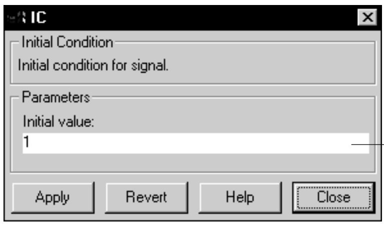
For example, these blocks illustrate how the IC block initializes a signal labeled “test signal.”



At t = 0, the signal value is 3. Afterwards, the signal value is 6.

The IC block is also useful in providing an initial guess for the algebraic state variables in the loop. For more information, see Chapter 10.

Dialog Box



The initial value for the signal. The default is 1.

Characteristics

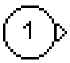
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Parameter only
States	0
Vectorized	Yes
Zero Crossing	No

Inport

Purpose Create an input port for a subsystem or an external input.

Library Connections

Description Inports are the links from outside a system into the system.



Simulink assigns Inport block port numbers according to these rules:

- It automatically numbers the Inport blocks within a top-level system or subsystem sequentially, starting with 1.
- If you add an Inport block, it is assigned the next available number.
- If you delete an Inport block, other port numbers are automatically renumbered to ensure that the Inport blocks are in sequence and that no numbers are omitted.
- If you copy an Inport block into a system, its port number is *not* renumbered unless its current number conflicts with an Inport block already in the system. If the copied Inport block port number is not in sequence, you must renumber the block or you will get an error message when you run the simulation or update the block diagram.

If the Inport block provides a vector signal, you can specify the width of the input to the Inport block as the **Port width** parameter or let Simulink determine it automatically by providing a value of -1 (the default).

The **Sample time** parameter is the rate at which the signal is coming into the system. The default (-1) causes the block to inherit its sample time from the block driving it. It may be appropriate to set this parameter for Inport blocks in the top-level system or in models where Inport blocks are driven by blocks whose sample time cannot be determined.

Inport Blocks in a Subsystem

Inport blocks in a subsystem represent inputs to the subsystem. A signal arriving at an input port on a Subsystem block flows out of the associated Inport block in that subsystem. The Inport block associated with an input port on a Subsystem block is the block whose **Port number** parameter matches the relative position of the input port on the Subsystem block. For example, the Inport block whose **Port number** parameter is 1 gets its signal from the block connected to the top-most port on the Subsystem block.

If you renumber the **Port number** of an Inport block, the block becomes connected to a different input port, although the block continues to receive its signal from the same block outside the subsystem.

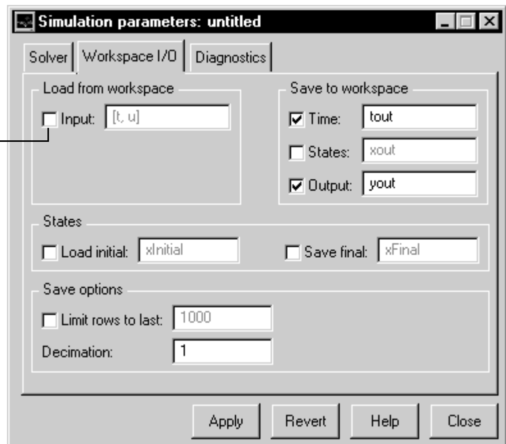
The Inport block name appears in the Subsystem block icon as a port label. To suppress display of the label, select the Inport block and choose **Hide Name** from the **Format** menu. Then, choose **Update Diagram** from the **Edit** menu.

Inport Blocks in a Top-Level System

Inport blocks in a top-level system have two uses: to supply external inputs from the workspace, which you can do by using either the **Simulation Parameters** dialog box or the `sim` command, and to provide a means for analysis functions to perturb the model:

- To supply external inputs from the workspace (using the **Simulation Parameters** dialog box). On the **Workspace I/O** tab, select the **Input** check box in the **Load from workspace** area. In the data field, specify either the matrix of time and input values, or a function that provides input values:

Select this check box to load input from the workspace using Inport blocks.



If you specify a matrix of time and input values, the first column is a vector of time values. Remaining columns are data at those time values, where each column supplies data for an Inport block, in port number order. If necessary, Simulink interpolates input data for time values not in the time vector.

If you specify a function, the function is evaluated at each time step to generate one or more inputs.

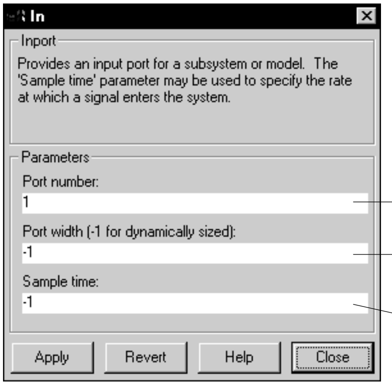
- To supply external inputs from the workspace (using the `sim` command). You can read data from the workspace using the `ut` argument:

```
[t,x,y] = sim('system', timespan, options, ut);
```

  If the system has more than one Inport block, `ut` is a matrix or function, as described in the preceding point.

- To provide a means for perturbation of the model by the `linmod` and `trim` analysis functions. Inport blocks define the points where inputs are injected into the system. For information about using Inport blocks with analysis commands, see Chapter 5.

Parameters and Dialog Box



The port number of the Inport block.

The width of the input signal to the Inport. Specify -1 to have it automatically determined.

The rate at which the signal is coming into the system.

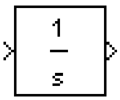
Characteristics

Sample Time	Inherited from driving block
Vectorized	Yes

Purpose Integrate a signal.

Library Linear

Description The Integrator block integrates its input. The output of the integrator is simply its state, the integral. The Integrator block allows you to:



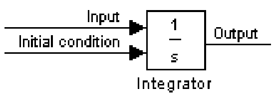
- Define initial conditions on the block dialog box or as input to the block.
- Output the state.
- Define upper and lower limits on the integral.
- Reset the state depending on an additional reset input.

Use the Discrete-Time Integrator block, described on page 9–42, when constructing a purely discrete system.

Defining Initial Conditions

You can define the initial conditions as a parameter on the block dialog box or input them from an external signal:

- To define the initial conditions as a block parameter, specify the **Initial condition source** parameter as **internal** and enter the value in the **Initial condition** parameter field.
- To provide the initial conditions from an external source, specify the **Initial condition source** parameter as **external**. An additional input port appears under the block input, as shown in this figure:

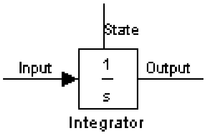


Using the State Port

In two known situations, you must use the state port instead of the output port:

- When the output of the block is fed back into the block through the reset port or the initial condition port, causing an algebraic loop. For an example of this situation, see the `bounce` model.
- When you want to pass the state from one conditionally executed subsystem to another, which may cause timing problems. For an example of this situation, see the `clutch` model.

You can correct these problems by passing the state through the state port rather than the output port. Although the values are the same, Simulink generates them at slightly different times, which protects your model from these problems.You output the block state by selecting the **Show state port** check box. By default, the state port appears on the top of the block, as shown in this figure:

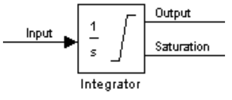


Limiting the Integral

To prevent the output from exceeding specifiable levels, select the **Limit output** check box and enter the limits in the appropriate parameter fields. Doing so causes the block to function as a limited integrator. When the output is outside the limits, the integral action is turned off to prevent integral wind up. During a simulation, you can change the limits but you cannot change whether the output is limited. The output is determined as follows:

- When the integral is less than the **Lower saturation limit** and the input is negative, the output is held at the **Lower saturation limit**.
- When the integral is between the **Lower saturation limit** and the **Upper saturation limit**, the output is the integral.
- When the integral is greater than the **Upper saturation limit** and the input is positive, the output is held at the **Upper saturation limit**.

To generate a signal that indicates when the state is being limited, select the **Show saturation port** check box. A saturation port appears below the block output port, as shown on this figure:



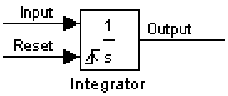
The signal has one of three values:

- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- -1 indicates that the lower limit is being applied.

When this option is selected, the block has three zero crossings: one to detect when it enters the upper saturation limit, one to detect when it enters the lower saturation limit, and one to detect when it leaves saturation.

Resetting the State

The block can reset its state to the specified initial condition based on an external signal. To cause the block to reset its state, select one of the **External reset** choices. A trigger port appears below the block's input port and indicates the trigger type, as shown in this figure.



Select **rising** to trigger the state reset when the reset signal has a rising edge. Select **falling** to trigger the state reset when the reset signal has a falling edge. Select **either** to trigger the reset when either a rising or falling signal occurs.

The reset port has direct feedthrough. If the block output is fed back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results. To resolve this loop, feed the block state into the reset port instead. To access the block's state, select the **Show state port** check box.

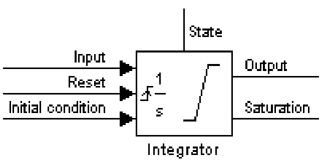
Specifying the Absolute Tolerance for the Block State

When your model contains states having vastly different magnitudes, defining the absolute tolerance for the model might not provide sufficient error control. To define the absolute tolerance for an Integrator block's state, provide a value for the **Absolute tolerance** parameter. If the block has more than one state, the same value is applied to all states.

For more information about error control, see "Error Tolerances" on page 4–11.

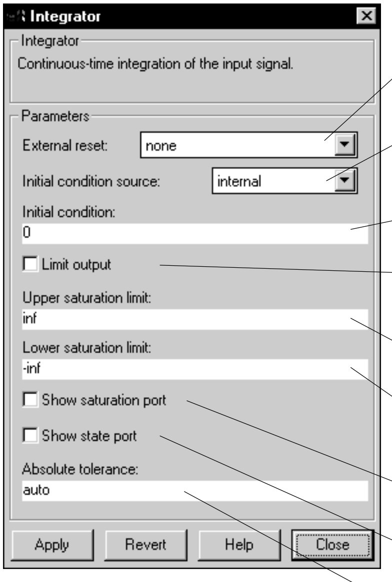
Choosing All Options

When all options are selected, the icon looks like this:



Integrator

Parameters and Dialog Box



Resets the states to their initial conditions when a trigger event (**rising**, **falling**, or **either**) occurs in the reset signal.

Gets the states' initial conditions from the **Initial condition** parameter (if set to **internal**) or from an external block (if set to **external**).

The states' initial conditions. Set the **Initial condition source** parameter value to **internal**.

If checked, limits the states to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

The upper limit for the integral. The default is `inf`.

The lower limit for the integral. The default is `-inf`.

If checked, adds a saturation output port to the block.

If checked, adds an output port to the block for the block's state.

Absolute tolerance for the block's states.

Characteristics

Direct Feedthrough	Yes, of the reset and external initial condition source ports
Sample Time	Continuous
Scalar Expansion	Of parameters
States	Inherited from driving block or parameter
Vectorized	Yes
Zero Crossing	If the **Limit output** option is selected, one for detecting reset; one each to detect upper and lower saturation limits, one when leaving saturation

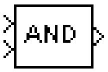
Purpose

Perform the specified logical operation on the input.

Library

Nonlinear

Description

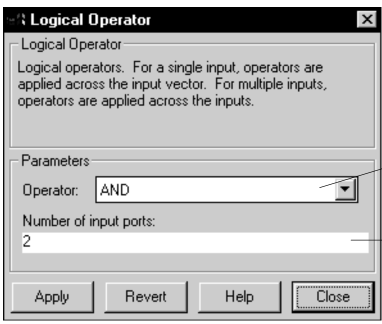


The Logical Operator block performs any of these logical operations on its inputs: AND, OR, NAND, NOR, XOR, and NOT. A nonzero input is treated as TRUE (`1`), a zero input as FALSE (`0`). The output depends on the number of inputs, their vector size, and the selected operator. The output is `1` if TRUE and `0` if FALSE. The block icon shows the selected operator.

- For two or more inputs, the block performs the operation between all of the inputs. If the inputs are vectors, the operation is performed between corresponding elements of the vectors to produce a vector output.
- For a single vector input, the block applies the operation (except the NOT operator) to all elements of that vector. The NOT operator accepts only one input, which can be a scalar or vector. If the input is a vector, the output is a vector of the same size containing the logical complements of the elements of the input vector.

When configured as a multi-input XOR gate, this block performs an addition modulo two operation as mandated by the IEEE Standard for Logic Elements.

Parameters and Dialog Box



The logical operator to be applied to the block inputs. Valid choices are the operators listed above.

The number of block inputs. The value must be appropriate for the selected operator.

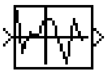
Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of inputs
Vectorized	Yes
Zero Crossing	No

Purpose Perform piecewise linear mapping of the input.

Library Nonlinear

Description The Look-Up Table block maps an input to an output using linear interpolation of the values defined in the block's parameters.



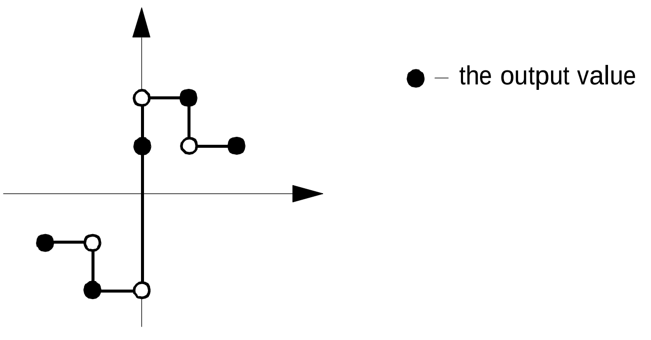
You define the table by specifying (either as row or column vectors) the **Vector of input values** and **Vector of output values** parameters. The block produces an output value by comparing the block input with values in the input vector:

- If it finds a value that matches the block's input, the output is the corresponding element in the output vector.
- If it does not find a value that matches, it performs linear interpolation between the two appropriate elements of the table to determine an output value. If the block input is less than the first or greater than the last input vector element, the block extrapolates using the first two or the last two points.

To map two inputs to an output, use the Look-Up Table (2-D) block. For more information, see page 9–89.

To create a table with step transitions, repeat an input value with different output values. For example, these input and output parameter values create the input/output relationship described by the plot that follows:

```
Vector of input values:   [–2 –1 –1  0 0 0 1 1 2]
Vector of output values:  [–1 –1 –2 –2 1 2 2 1 1]
```



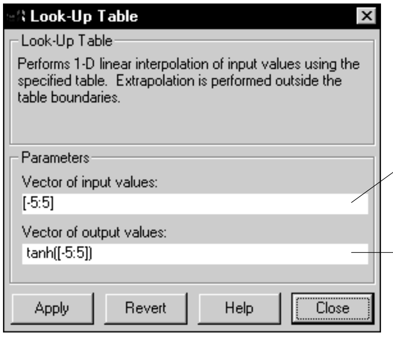
This example has three step discontinuities: at u = -1, 0, and +1.

When there are two points at a given input value, the block generates output according to these rules:

- When *u* is less than zero, the output is the value connected with the point first encountered when moving away from the origin in a negative direction. In this example, when *u* is -1, *y* is -2, marked with a solid circle.
- When *u* is greater than zero, the output is the value connected with the point first encountered when moving away from the origin in a positive direction. In this example, when *u* is 1, *y* is 2, marked with a solid circle.
- When *u* is at the origin and there are two output values specified for zero input, the actual output is their average. In this example, if there were no point at *u* = 0 and *y* = 1, the output would be 0, the average of the two points at *u* = 0. If there are three points at zero, the block generates the output associated with the middle point. In this example, the output at the origin is 1.

The Look-Up Table block icon displays a graph of the input vector versus the output vector. When a parameter is changed on the block's dialog box, the graph is automatically redrawn when you press the **Apply** or **Close** button.

Parameters and Dialog Box



The vector of values containing possible block input values. This vector must be the same size as the output vector. The input vector must be monotonically increasing.

The vector of values containing block output values. This vector must be the same size as the input vector.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Vectorized	Yes
Zero Crossing	No

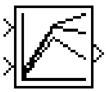
Purpose

Perform piecewise linear mapping of two inputs.

Library

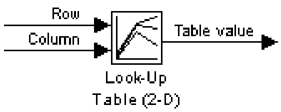
Nonlinear

Description



The Look-Up Table (2-D) block maps the block inputs to an output using linear interpolation of a table of values defined by the block's parameters.

You define the possible output values as the **Table** parameter. You define the values that correspond to its rows and columns with the **Row** and **Column** parameters. The block generates an output value by comparing the block inputs with the **Row** and the **Column** parameters. The first input identifies a row, and the second input identifies a column, as shown by this figure:



The block generates output based on the input values:

- If the inputs match row and column parameter values, the output is the table value at the intersection of the row and column.
- If the inputs do not match row and column parameter values, the block generates output by linearly interpolating between the appropriate table values. If either or both block inputs are less than the first or greater than the last row or column parameter values, the block extrapolates from the first two or last two points.

If either the **Row** or **Column** parameter has a repeating value, the block chooses a value using the technique described for the Look-Up Table block.

The Look-Up Table block allows you to map a single input value into a vector of output values. That block is described on page 9–87.

Example

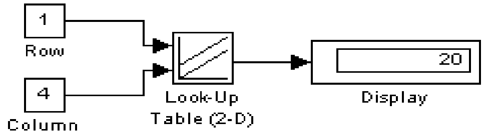
In this example, the block parameters are defined as:

```
Row:       [1 2]
Column:    [3 4]
Table:     [10 20; 30 40]
```

The first figure shows the block outputting a value at the intersection of block inputs that match row and column values. The first input is 1 and the second

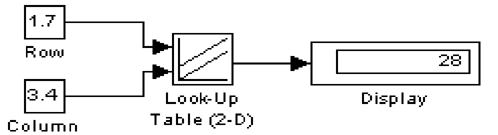
input is 4. These values select the table value at the intersection of the first row (row parameter value 1) and second column (column parameter value 4):

	3	4
1	10	20
2	30	40

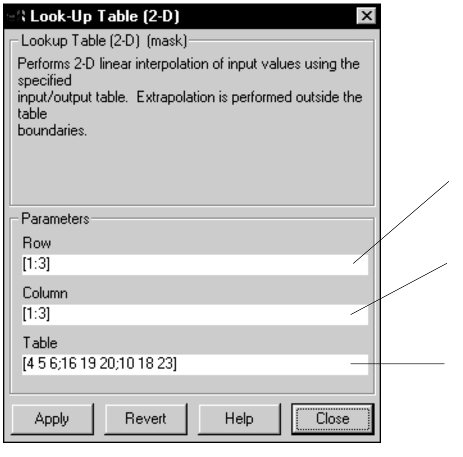


In the second figure, the first input is 1.7 and the second is 3.4. These values cause the block to interpolate between row and column values, as shown in the table at the left. The value at the intersection (28) is the output value.

	3	3.4	4
1	10	14	20
1.7	24	28	34
2	30	34	40



Parameters and Dialog Box



The row values for the table, entered as a vector. The vector values must increase monotonically.

The column values for the table, entered as a vector. The vector values must increase monotonically.

The table of output values. The matrix size must match the dimensions defined by the **Row** and **Column** parameters.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving blocks
Scalar Expansion	Of one input if the other is a vector
Vectorized	Yes
Zero Crossing	No

Manual Switch

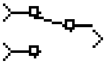
Purpose

Switch between two inputs.

Library

Nonlinear

Description



The Manual Switch block is a toggle switch that selects one of its two inputs to pass through to the output. To toggle between inputs, double-click on the block icon (there is no dialog box). The selected input is propagated to the output, while the unselected input is discarded. You can set the switch before the simulation is started or throw it while the simulation is executing to interactiely control the signal flow.

The Manual Switch block accepts all input types and retains its current state when the model is saved.

Parameters and Dialog Box

None

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Vectorized	Yes
Zero Crossing	No

Purpose

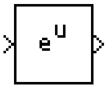
Perform a mathematical function.

Library

Nonlinear

Description

The Math Function block performs numerous common mathematical functions.

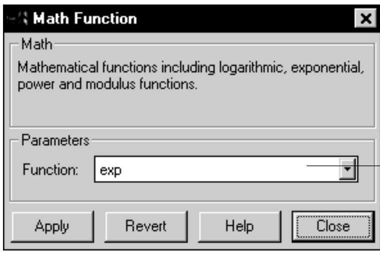


You can select one of these functions from the **Function** list: `exp`, `log`, `10`^{u}, `log10`, `square`, `sqrt`, `pow`, `reciprocal`, `hypot`, `rem`, and `mod`. The block output is the result of the function operating on the input or inputs.

The name of the function appears on the block icon. Simulink automatically draws the appropriate number of input ports.

Use the Math Function block instead of the Fcn block when you want vectorized output because the Fcn block can produce only scalar output.

Parameters and Dialog Box



The mathematical function.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of the input when the function requires two inputs
Vectorized	Yes
Zero Crossing	No

MATLAB Fcn

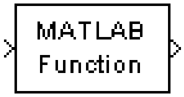
Purpose

Apply a MATLAB function or expression to the input.

Library

Nonlinear

Description



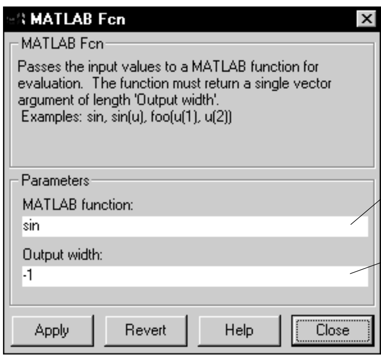
The MATLAB Fcn block applies the specified MATLAB function or expression to the input. The block accepts one input and generates one output. The specified function or expression is applied to the input. The output of the function must match the output width of the block or an error occurs.

Here are some sample valid expressions for this block:

```
sin
atan2(u(1), u(2))
u(1)^u(2)
```

NOTE This block is slower than the Fcn block because it calls the MATLAB parser during each integration step. Consider using built-in blocks (such as the Fcn block or the Math Function block) instead, or writing the function as an M-file or MEX-file S-function, then accessing it using the S-Function block.

Parameters and Dialog Box



The function or expression. If you specify a function only, it is not necessary to include the input argument in parentheses.

The output width. If the output width is to be the same as the input width, specify -1. Otherwise, you must specify the correct width or an error will result.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Vectorized	Yes
Zero Crossing	No

Matrix Gain

Purpose

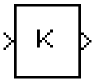
Multiply the input by a matrix.

Library

Linear

Description

The Matrix Gain block implements a matrix gain. It generates its output by multiplying its vector input by a specified matrix:



$$y = Ku$$

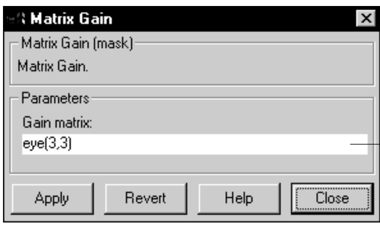
where K is the gain and u is the input.

If the specified matrix has `m` rows and `n` columns, then the input to this block should be a vector of length `n`. The output is a vector of length `m`.

The block icon always displays `K`.

If the matrix contains zeros, Simulink converts the matrix gain to a sparse matrix for efficient multiplication.

Parameters and Dialog Box



The gain, specified as a matrix. The default is `eye(3,3)`.

Characteristics

Direct Feedthrough	Yes
Sample Time	Continuous
Scalar Expansion	No
States	0
Vectorized	Yes
Zero Crossing	No

Purpose

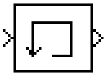
Output the block input from the previous integration step.

Library

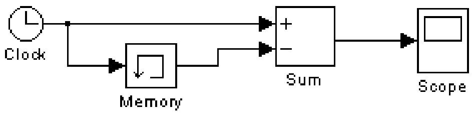
Nonlinear

Description

The Memory block outputs its input from the previous time step, applying a one integration step sample-and-hold to its input signal.

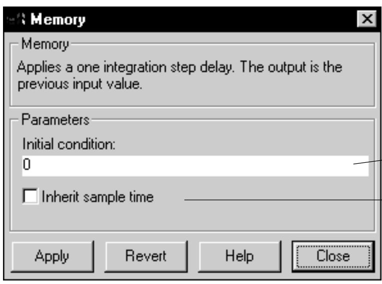


This sample model (which, to provide more useful information, would be part of a larger model) demonstrates how to display the step size used in a simulation. The Sum block subtracts the time at the previous step, generated by the Memory block, from the current time, generated by the clock.



NOTE Avoid using the Memory block when integrating with `ode15s` or `ode113`, unless the input to the block does not change.

Parameters and Dialog Box



The output at the initial integration step.

Check this box to cause the sample time to be inherited from the driving block.

Characteristics

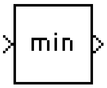
Direct Feedthrough	No
Sample Time	Continuous, but inherited if the **Inherit sample time** check box is selected
Scalar Expansion	Of the **Initial condition** parameter
Vectorized	Yes
Zero Crossing	No

MinMax

Purpose Output the minimum or maximum input value.

Library Nonlinear

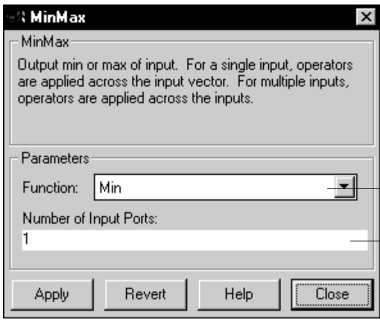
Description The MinMax block outputs either the minimum or the maximum element or elements of the input(s). You can choose which function to apply by selecting one of the choices from the **Function** parameter list.



If the block has one input port, the block outputs a scalar that is the minimum or maximum element of the input vector.

If the block has more than one input port, the block performs an element-by-element comparison of the input vectors. Each element of the block output vector is the result of the comparison of the elements of the input vectors.

Parameters and Dialog Box



The function (`min` or `max`) to apply to the input.

The number of inputs to the block.

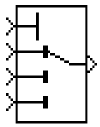
Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block
Scalar Expansion	Of the inputs
Vectorized	Yes
Zero Crossing	Yes, to detect minimum and maximum values

Purpose Choose between block inputs.

Library Nonlinear

Description The Multiport Switch block chooses between a number of inputs.



The first (top) input is the control input and the other inputs are the switch inputs. The value of the control input determines which switch input to pass through to the output port. Simulink rounds the control input value to the nearest (positive) integer and passes the switch input that corresponds to that value, according to this table:

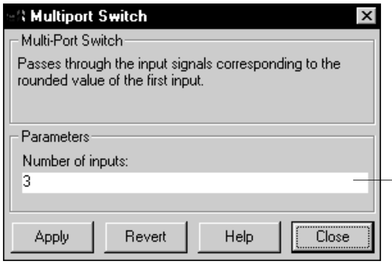
This control input value	Passes this switch input
Up to 1.4999...	First
From 1.5 to 2.4999...	Second
From 2.5 to 3.4999...	Third
etc.	etc.

Switch inputs can be scalar or vector. The control input can be a scalar or a vector. The block output is determined by these rules:

- If inputs are scalar, the output is a scalar.
- If the block has more than one switch input, at least one of which is a vector, the output is a vector. Any scalar inputs are expanded to vectors.
- If the block has only one switch input and that input is a vector, the block output is the element of the vector that corresponds to the rounded value of the control input.

Multiport Switch

Parameters and Dialog Box



The number of switch inputs to the block.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block(s)
Scalar Expansion	Yes
Vectorized	Yes
Zero Crossing	No

Purpose

Combine several input lines into a vector line.

Library

Connections

Description

The Mux block combines several input lines into one vector line. Each input line can carry a scalar or vector signal. The output of a Mux block is a vector.

If you define the **Number of inputs** parameter as a scalar, Simulink determines the input widths by checking the output ports of the blocks feeding the Mux block. If any input is a vector, all of its elements are combined by the block.

If it is necessary to define input widths explicitly, you can specify them as a vector. Include elements with `–1` values for those inputs whose widths are to be determined dynamically (during the simulation). If an input signal width does not match the expected width, Simulink displays an error message.

For example, `[4 1 2]` indicates three inputs forming a seven-element output vector: the first four output elements are from the first input, the fifth element comes from the second input, and the sixth and seventh elements come from the third input. If it is not important that these inputs have fixed widths, you could specify the **Number of inputs** as 3.

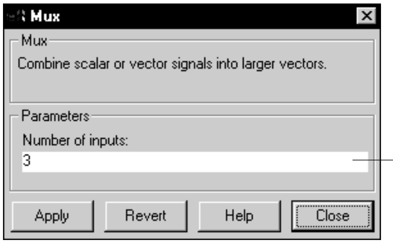
To specify three inputs where the first input vector must have four elements, you could specify `[4 –1 –1]`. Simulink determines the widths of the second and third inputs and sizes the output width accordingly.

Simulink draws the Mux block with the specified number of inputs. If you change the number of input ports, Simulink adds or removes them from the bottom of the block icon.

Using a Variable to Provide the Number of Inputs Parameter

When you specify the **Number of inputs** parameter as a variable, Simulink issues an error message if the variable is undefined in the workspace.

Parameters and Dialog Box



The number and width of inputs. The total of the input widths must match the width of the output line.

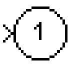
Characteristics

Sample Time	Inherited from driving block(s)
Vectorized	Yes

Purpose Create an output port for a subsystem or an external output.

Library Connections

Description Outports are the links from a system to a destination outside the system.



Simulink assigns Outport block port numbers according to these rules:

- It automatically numbers the Outport blocks within a top-level system or subsystem sequentially, starting with 1.
- If you add an Outport block, it is assigned the next available number.
- If you delete an Outport block, other port numbers are automatically renumbered to ensure that the Outport blocks are in sequence and that no numbers are omitted.
- If you copy an Outport block into a system, its port number is *not* renumbered unless its current number conflicts with an Outport block already in the system. If the copied Outport block port number is not in sequence, you must renumber the block or you will get an error message when you run the simulation or update the block diagram.

Outport Blocks in a Subsystem

Outport blocks in a subsystem represent outputs from the subsystem. A signal arriving at an Outport block in a subsystem flows out of the associated output port on that Subsystem block. The Outport block associated with an output port on a Subsystem block is the block whose **Port number** parameter matches the relative position of the output port on the Subsystem block. For example, the Outport block whose **Port number** parameter is 1 sends its signal to the block connected to the top-most output port on the Subsystem block.

If you renumber the **Port number** of an Outport block, the block becomes connected to a different output port, although the block continues to send the signal to the same block outside the subsystem.

When you create a subsystem by selecting existing blocks, if more than one Outport block is included in the grouped blocks, Simulink automatically renumbers the ports on the blocks.

The Outport block name appears in the Subsystem block icon as a port label. To suppress display of the label, select the Outport block and choose **Hide Name** from the **Format** menu.

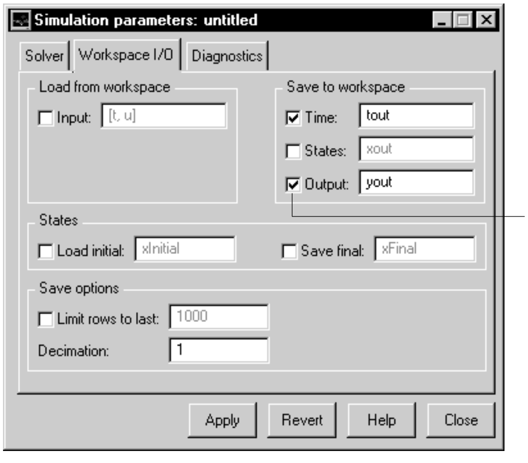
Outport Blocks in a Conditionally Executed Subsystem

When an Outport block is in a triggered and/or enabled subsystem, you can specify what happens to its output when the subsystem is disabled: it can be **reset** to an initial value or **held** at its most recent value. The **Output when disabled** popup menu provides these options. The **Initial output** parameter is the value of the output before the subsystem executes and, if the **reset** option is chosen, while the subsystem is disabled.

Outport Blocks in a Top-Level System

Outport blocks in a top-level system have two uses: to supply external outputs to the workspace, which you can do by using either the **Simulation Parameters** dialog box or the `sim` command, and to provide a means for analysis functions to obtain output from the system.

- To supply external outputs to the workspace (using the **Simulation Parameters** dialog box). On the **Workspace I/O** tab, select the **Output** check box in the **Save to workspace** area and specify the variable in the data field:



Select this check box to write output to the workspace using Outport blocks.

  If you specify one variable name in the edit field, all output is written to that variable. If the system has more than one Outport block, the variable is a matrix, where each column contains data for a different Outport block. The column order matches the order of the port numbers for the Outport blocks. If you specify more than one variable name, data from each Outport block is written to a different variable. For example, if the system has two Outport

blocks, to save data from Outport block 1 to `speed` and the data from Outport block 2 to `dist`, you specify `speed,dist` in the **Output** field.

- To supply external outputs to the workspace (using the `sim` command). You can write output to the workspace using the third return argument:

```
[t,x,y] = sim(...);
```

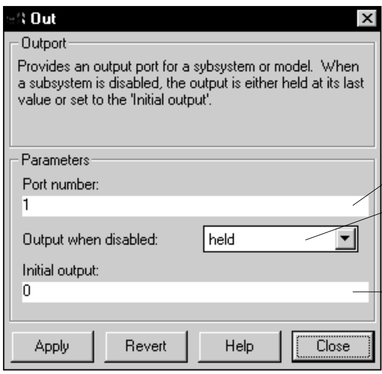
If the system has more than one Outport block, the above command writes `y` as a matrix, with each column containing data for a different Outport block. The column order matches the order of the port numbers for the Outport blocks.

If you specify more than one variable name after the second (state) argument, data from each Outport block is written to a different variable. For example, if the system has two Outport blocks, to save data from Outport block 1 to `speed` and the data from Outport block 2 to `dist`, you could specify this command:

```
[t,x,speed,dist] = sim(...);
```

- To provide a means for the `linmod` and `trim` analysis functions to obtain output from the system. For more information about using Outports with analysis commands, see Chapter 5.

Parameters and Dialog Box



The port number of the Outport block.

For conditionally executed subsystems, what happens to the block output when the system is disabled.

For conditionally executed subsystems, the block output before the subsystem executes and while it is disabled.

Characteristics

Sample Time	Inherited from driving block
Vectorized	Yes

Product

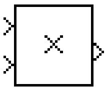
Purpose

Generate the product or quotient of block inputs.

Library

Nonlinear

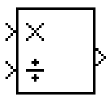
Description



The Product block multiplies or divides block inputs, depending on the value of the **Number of inputs** parameter:

- If the value is a combination of * and / symbols, the number of block inputs is equal to the number of symbols. The block icon shows the appropriate symbol adjacent to each input port. The block output is the product of all inputs marked * divided by all inputs marked /.

  For example, this block icon is the result of entering */ as the parameter value:

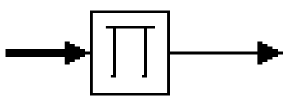
- If the value is a scalar greater than 1, the block multiplies all inputs. If any input is a vector, the block output is an element-by-element product across the inputs. If all inputs are scalars, the output is a scalar. For a block having n inputs, if any input is a vector, each element of the output is generated as:

  $$y_i = u1_i \times u2_i \times \ldots \times un_i$$

- If the value is 1, the block output is the scalar product of the elements of the input vector:

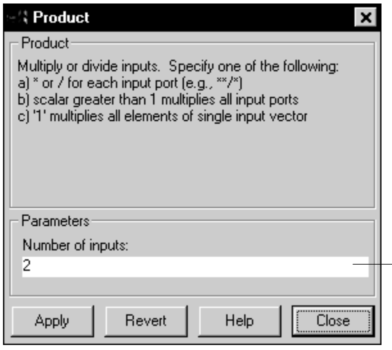
  $$y = \Pi u_i$$

  This model represents using the Product block in this way. The solid line input signal indicates that the input is a vector:

If necessary, Simulink resizes the block to show all input ports. If the number of inputs is changed, ports are added or deleted from the bottom of the block.

Parameters and Dialog Box



Either the number of inputs to the block or a combination of * and / symbols. The default is 2.

Characteristics

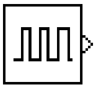
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Vectorized	Yes
Zero Crossing	No

Pulse Generator

Purpose Generate pulses at regular intervals.

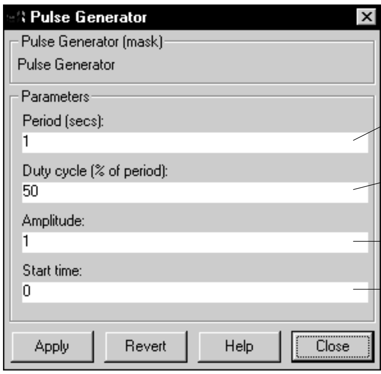
Library Sources

Description The Pulse Generator block generates a series of pulses at regular intervals.



Use the Pulse Generator block for continuous systems. To generate discrete signals, use the Discrete Pulse Generator block, described on page 9–39.

Parameters and Dialog Box



The pulse period in seconds. The default is one second.

The duty cycle: the percentage of the pulse period that the signal is on. The default is 50 percent.

The pulse amplitude. The default is 1.

The delay before the pulse is generated, in seconds. The default is 0 seconds.

Characteristics

Sample Time	Inherited
Scalar Expansion	Of parameters
Vectorized	Yes
Zero Crossing	No

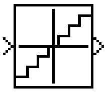
Purpose

Discretize input at a specified interval.

Library

Nonlinear

Description

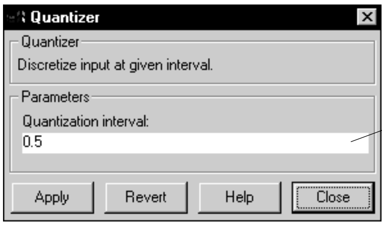


The Quantizer block passes its input signal through a stair-step function so that many neighboring points on the input axis are mapped to one point on the output axis. The effect is to quantize a smooth signal into a stair-step output. The output is computed using the round-to-nearest method, which produces an output that is symmetric about zero:

```
y = q * round(u/q)
```

where y is the output, u the input, and q the **Quantization interval** parameter.

Parameters and Dialog Box



The interval around which the output is quantized. Permissible output values for the Quantizer block are n*q, where n is an integer and q the **Quantization interval**. The default is 0.5.

Characteristics

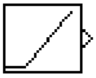
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of parameter
Vectorized	Yes
Zero Crossing	No

Ramp

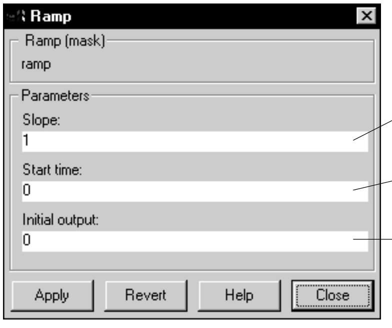
Purpose Generate constantly increasing or decreasing signal.

Library Sources

Description The Ramp block generates a signal that starts at a specified time and value and changes by a specified rate.



Parameters and Dialog Box



The rate of change of the generated signal. The default is 1.

The time at which the signal begins to be generated. The default is 0.

The initial value of the signal. The default is 0.

Characteristics

Sample Time	Inherited from driven block
Scalar Expansion	Yes
Vectorized	Yes
Zero Crossing	Yes

Purpose

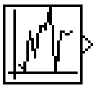
Generate normally distributed random numbers.

Library

Sources

Description

The Random Number block generates normally distributed random numbers. The seed is reset to the specified value each time a simulation starts.

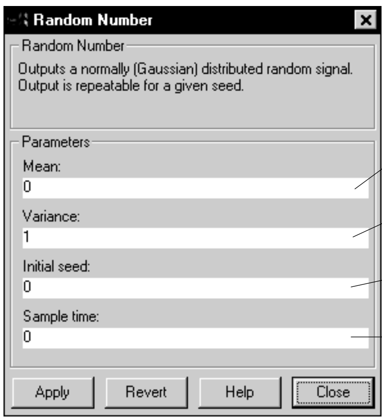


By default, the sequence produced has a mean of 0 and a variance of 1, although you can vary these parameters. The sequence of numbers is repeatable and can be produced by any Random Number block with the same seed and parameters. To generate a vector of random numbers with the same mean and variance, specify the **Initial seed** parameter as a vector.

To generate uniformly distributed random numbers, use the Uniform Random Number block, described on page 9–158.

Avoid integrating a random signal because solvers are meant to integrate relatively smooth signals. Instead, use the Band-Limited White Noise block.

Parameters and Dialog Box



The mean of the random numbers. The default is 0.

The variance of the random numbers. The default is 1.

The starting seed for the random number generator. The default is 0.

The time interval between samples. The default is 0, causing the block to have continuous sample time.

Characteristics

Sample Time	Continuous or discrete
Scalar Expansion	Of parameters
Vectorized	Yes
Zero Crossing	No

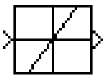
Purpose

Limit the rate of change of a signal.

Library

Nonlinear

Description



The Rate Limiter block limits the first derivative of the signal passing through it. The output changes no faster than the specified limit. The derivative is calculated using this equation:

$$rate = \frac{u(i) - y(i-1)}{t(i) - t(i-1)}$$

u(i) and *t(i)* are the current block input and time, and *y(i–1)* and *t(i–1)* are the output and time at the previous step. The output is determined by comparing *rate* to the **Rising slew rate** and **Falling slew rate** parameters:

- If *rate* is greater than the **Rising slew rate** parameter (R), the output is calculated as:

  $$y(i) = \Delta t \cdot R + y(i-1)$$

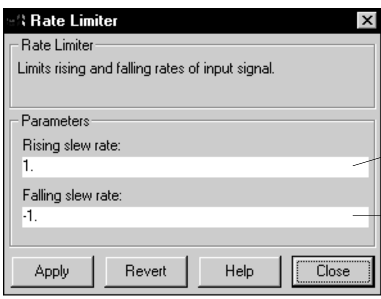
- If *rate* is less than the **Falling slew rate** parameter (F), the output is calculated as:

  $$y(i) = \Delta t \cdot F + y(i-1)$$

- If *rate* is between the bounds of R and F, the change in output is equal to the change in input:

  $$y(i) = u(i)$$

Parameters and Dialog Box



The limit of the derivative of an increasing input signal.

The limit of the derivative of a decreasing input signal.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of input and parameters
Vectorized	Yes
Zero Crossing	No

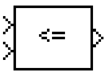
Purpose

Perform the specified relational operation on the input.

Library

Nonlinear

Description



The Relational Operator block performs a relational operation on its two inputs and produces output according to the following table:

Operator	Output
==	TRUE if the first input is equal to the second input
!=	TRUE if the first input is not equal to the second input
<	TRUE if the first input is less than the second input
<=	TRUE if the first input is less than or equal to the second input
>=	TRUE if the first input is greater than or equal to the second input
>	TRUE if the first input is greater than the second input

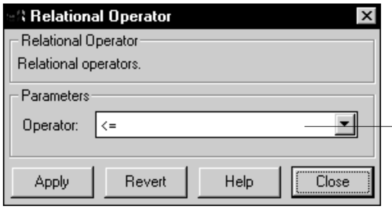
If the result is TRUE, the output is 1; if FALSE, it is 0. You can specify inputs as scalars, vectors, or a combination of a scalar and a vector:

- For scalar inputs, the output is a scalar.
- For vector inputs, the output is a vector, where each element is the result of an element-by-element comparison of the input vectors.
- For mixed scalar/vector inputs, the output is a vector, where each element is the result of a comparison between the scalar and the corresponding vector element.

The block icon displays the selected operator.

Relational Operator

Parameters and Dialog Box



The relational operator to be applied to the block inputs.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of inputs
Vectorized	Yes
Zero Crossing	Yes, to detect when the output changes

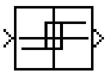
Purpose

Switch output between two constants.

Library

Nonlinear

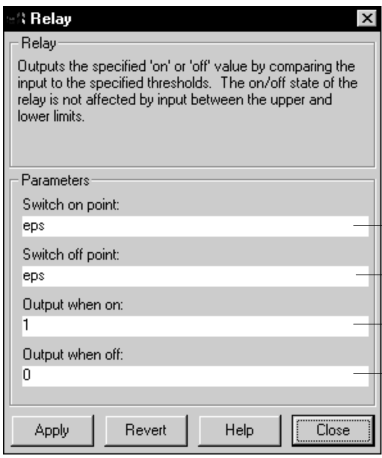
Description



The Relay block allows the output to switch between two specified values. When the relay is on, it remains on until the input drops below the value of the **Switch off point** parameter. When the relay is off, it remains off until the input exceeds the value of the **Switch on point** parameter. The block accepts one input and generates one output.

Specifying a **Switch on point** value greater than the **Switch off point** value models hysteresis, whereas specifying equal values models a switch with a threshold at that value. The **Switch on point** value must be greater than or equal to the **Switch off point**.

Parameters and Dialog Box



The on threshold for the relay. The default is eps.

The off threshold for the relay. The default is eps.

The output when the relay is on. The default is 1.

The output when the relay is off. The default is 0.

Characteristics

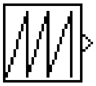
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Vectorized	Yes
Zero Crossing	Yes, to detect switch on and switch off points

Repeating Sequence

Purpose Generate a repeatable arbitrary signal.

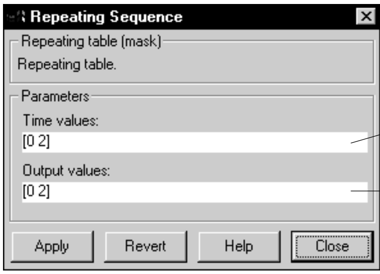
Library Sources

Description The Repeating Sequence block allows you to specify an arbitrary signal to be repeated regularly over time. When the simulation reaches the maximum time value in the **Time values** vector, the signal is repeated.



This block is implemented using the one-dimensional Look-Up Table block, performing linear interpolation between points.

Parameters and Dialog Box



A vector of monotonically increasing time values. The default is `[0 2]`.

A vector of output values. Each corresponds to the time value in the same column. The default is `[0 2]`.

Characteristics

Sample Time	Continuous
Scalar Expansion	No
Vectorized	No
Zero Crossing	No

Purpose

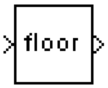
Perform a rounding function.

Library

Nonlinear

Description

The Rounding Function block performs common mathematical rounding functions.

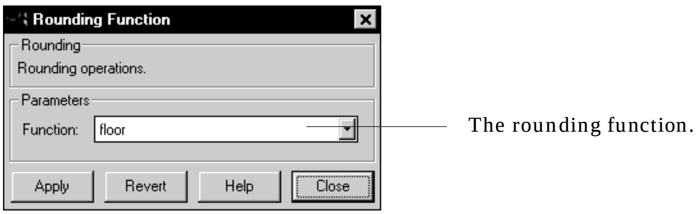


You can select one of these functions from the **Function** list: `floor`, `ceil`, `round`, and `fix`. The block output is the result of the function operating on the input or inputs.

The name of the function appears on the block icon.

Use the Rounding Function block instead of the Fcn block when you want vectorized output because the Fcn block can produce only scalar output.

Parameters and Dialog Box



Characteristics

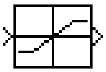
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Vectorized	Yes
Zero Crossing	No

Saturation

Purpose Limit the range of a signal.

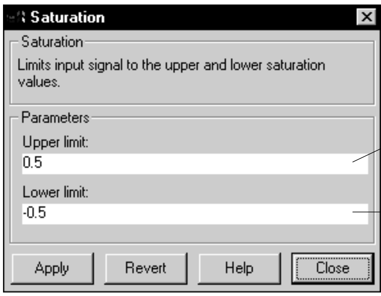
Library Nonlinear

Description The Saturation block imposes upper and lower bounds on a signal. When the input signal is within the range specified by the **Lower limit** and **Upper limit** parameters, the input signal passes through unchanged. When the input signal is outside these bounds, the signal is clipped to the upper or lower bound.



When the parameters are set to the same value, the block outputs that value.

Parameters and Dialog Box



The upper bound on the input signal. While the signal is above this value, the block output is set to this value.

The lower bound on the input signal. While the signal is below this value, the block output is set to this value.

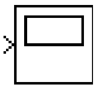
Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of parameters and input
Vectorized	Yes
Zero Crossing	Yes, to detect when the signal reaches a limit, and when it leaves the limit

Purpose Display signals generated during a simulation.

Library Sinks

Description

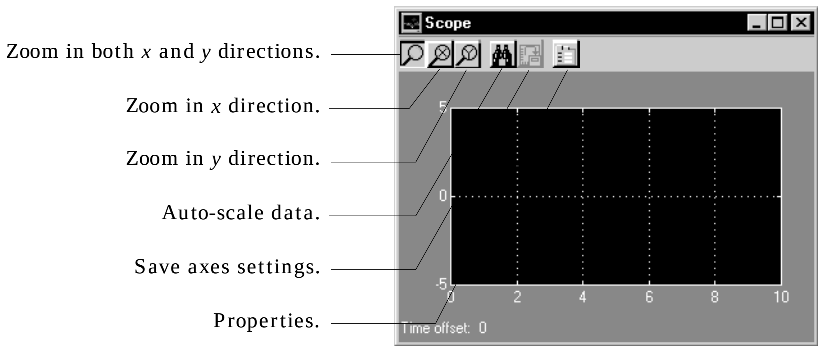


The Scope block displays its input with respect to simulation time. The block accepts one input and can display up to 30 signals. The Scope allows you to adjust the amount of time and the range of input values displayed. You can move and resize the Scope window and you can modify the Scope's parameter values during the simulation.

When you start a simulation, Simulink does not open Scope windows, although it does write data to connected Scopes. As a result, if you open a Scope after a simulation, the Scope's input signal or signals will be displayed.

If the signal is continuous, the Scope produces a point-to-point plot. If the signal is discrete, the Scope produces a stairstep plot.

The Scope provides toolbar buttons that enable you to zoom in on displayed data, display all the data input to the Scope, preserve axes settings from one simulation to the next, limit data displayed, and save data to the workspace. The toolbar buttons are labeled in this figure, which shows the Scope window as it appears when you open a Scope block:



Displaying Vector Signals

The Scope can display up to 30 signals. When more than one signal is displayed, the Scope uses different colors in this order: yellow, magenta, cyan, red, green, and dark blue. When more than six signals are displayed, the Scope cycles through the colors in the order listed above.

Using the Scope as a Floating Scope

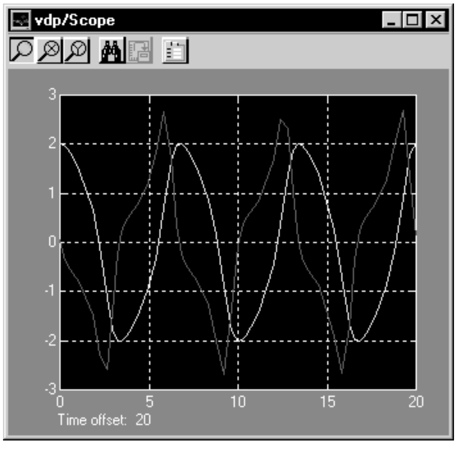
A floating Scope is a Scope block that can display the signals carried on one or more lines.

To add a floating Scope to a model, copy a Scope block into the model window, then open the block. Select the **Properties** button on the block's toolbar (the right-most button). Then, select the **Settings** tab and select the **Floating scope** check box.

To use a floating Scope during a simulation, first open the block. To display the signals carried on a line, select the line. Hold down the **Shift** key while clicking on another line to select multiple lines. It may be necessary to press the **Auto-scale data** button on the Scope's toolbar to find the signal and adjust the axes to the signal values.

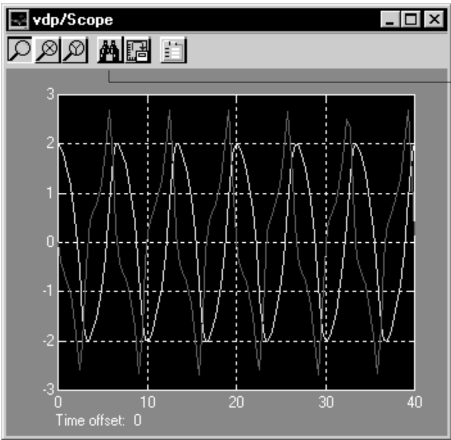
A model can contain more than one floating Scope, although it is not useful to have more than one floating Scope in a window.

This figure shows the Scope block displaying the output of the `vdp` model. The simulation was run for 40 seconds. Note that this scope shows the final 20 seconds of the simulation. The **Time offset** field displays the number of seconds on previous screens.



Auto-Scaling the Scope Axes

This figure shows the same output after pressing the **Auto-scale** toolbar button, which automatically scales both axes to display all stored simulation data. In this case, the *y*-axis was not scaled because it was already set to the appropriate limits.



The Auto-scale button

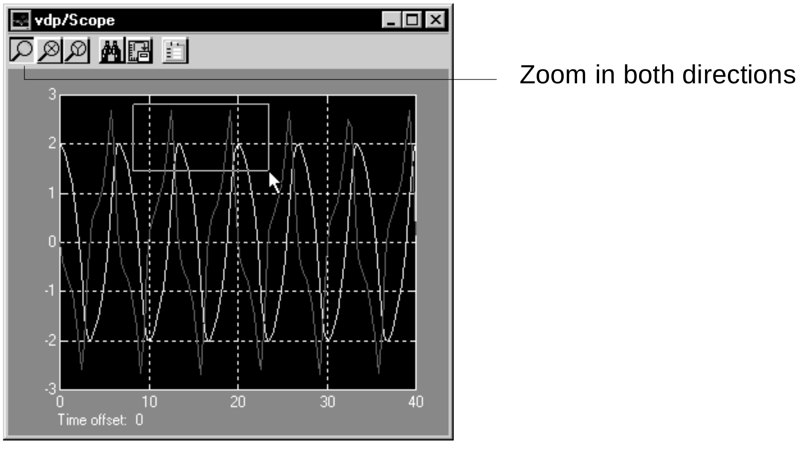
If you click on the **Auto-scale** button while the simulation is running, the axes are auto-scaled based on the data displayed on the current screen, and the auto-scale limits are saved as the defaults. This enables you to use the same limits for another simulation.

Zooming in on Data

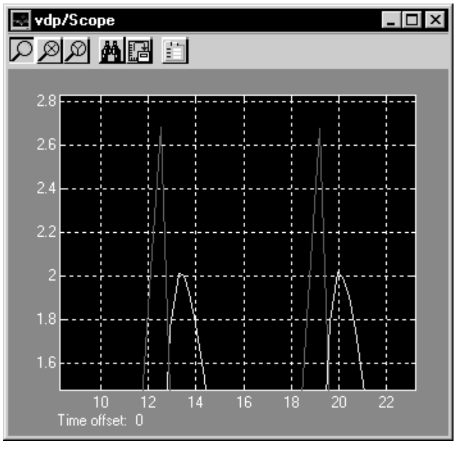
You can zoom in on data in both the *x* and *y* directions at the same time, or in either direction separately. The zoom feature is not active while the simulation is running.

To zoom in on data in both directions at the same time, make sure the left-most **Zoom** toolbar button is selected. Then, define the zoom region using a bounding box. When you release the mouse button, the Scope displays the data in that

area. This figure shows a region of the displayed data enclosed within a bounding box.

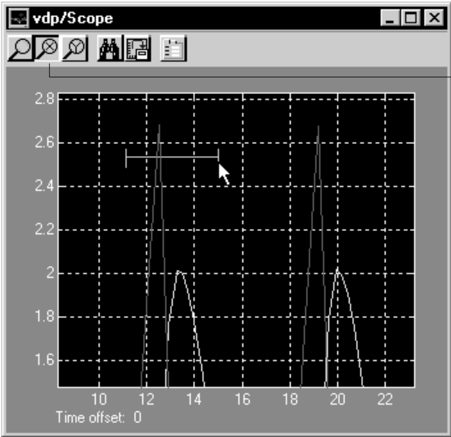


This figure shows the zoomed region, which appears after you release the mouse button.



To zoom in on data in just the *x* direction, click on the middle **Zoom** toolbar button. Define the zoom region by positioning the pointer at one end of the region, pressing and holding down the mouse button, then moving the pointer

to the other end of the region. This figure shows the Scope after defining the zoom region but before releasing the mouse button:



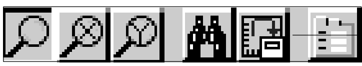
Zoom in x direction

When you release the mouse button, the Scope displays the magnified region.

Zooming in the *y* direction works the same way except that you press the right-most **Zoom** toolbar button before defining the zoom region.

Saving the Axes Settings

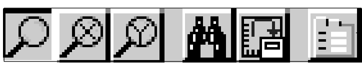
The **Save axes settings** toolbar button enables you to store the current *x*- and *y*-axis settings so you can apply them to the next simulation.



the Save axes settings button

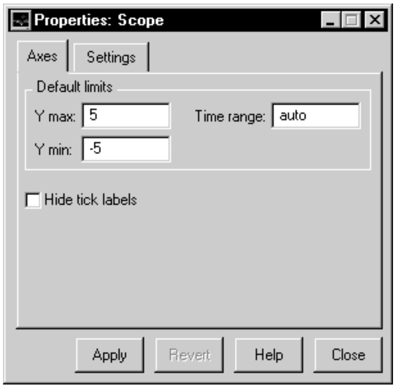
You might want to do this after zooming in on a region of the displayed data so you can see the same region in another simulation. The time range is inferred from the current *x*-axis limits.

You can also change axes limits by choosing the **Properties** toolbar button.



the Properties button

When you click on the **Properties** button, this dialog box appears:

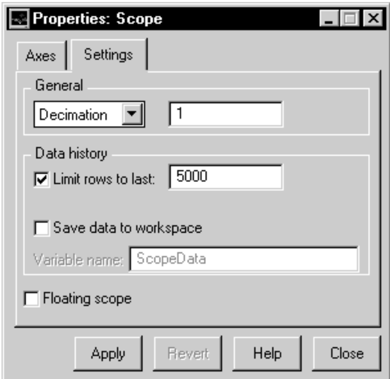


The dialog box has two tabs: **Axes** and **Settings**. You apply the current axes limits by clicking on the **Apply** or **Close** button. The values that appear in these fields are the values that will be used in the next simulation.

You can change the *y*-axis settings by modifying the values in the **Y max** and **Y min** fields. Change the *x*-axis limits by entering a number or `auto` in the **Time range** field. Entering a number of seconds causes each screen to display the amount of data that corresponds to that number of seconds. Enter `auto` to set the *x*-axis to the duration of the simulation. Apply the changes by clicking on the **Apply** or **Close** button. Do not enter variable names in these fields.

Controlling Data Collection and Display

You can control the amount of data that the Scope stores and displays by setting fields on the **Settings** tab:



To specify a decimation factor, enter a number in the data field to the right of the **Decimation** choice. To display data at a sampling interval, select the **Sample time** choice and enter a number in the data field.

You can limit the number of points by checking the **Limit points to last** check box and entering a value in its data field.

You can automatically save the data collected by the Scope at the end of the simulation by checking the **Save data to workspace** check box and entering a variable name in the **Variable name** field. The specified name must be unique among all data logging variables being used in the model. Other data logging variables are defined on other Scope blocks, To Workspace blocks, and simulation return variables such as time, states, and outputs. Being able to save Scope data to the workspace means that it is not necessary to send the same data stream to both a Scope block and a To Workspace block.

The Scope relies on its data history for zooming and auto-scaling operations. If the number of points is limited to 1,000 and the simulation generates 2,000 points, only the last 1,000 are available for regenerating the display.

Characteristics

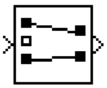
Sample Time	Inherited from driving block or settable
States	0

Selector

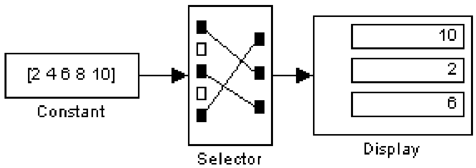
Purpose Select input elements.

Library Connections

Description The Selector block generates as output selective elements of the input vector.

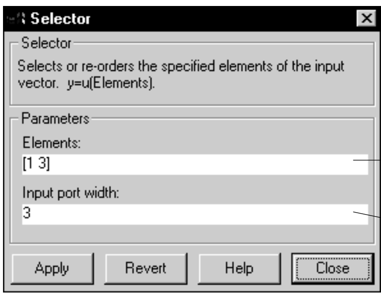


The **Elements** parameter defines the order of the input vector elements in the output vector. The parameter must be specified as a vector unless only one element is being selected. For example, this model shows the Selector block icon and the output for an input vector of `[2 4 6 8 10]` and an **Elements** parameter value of `[5 1 3]`:



The block icon displays the ordering of input vector elements graphically. If the block is not large enough, it displays the block name.

Parameters and Dialog Box



The order that the input elements are to appear in the output vector.

The number of elements in the input vector.

Characteristics

Sample Time	Inherited from driving block
Vectorized	Yes

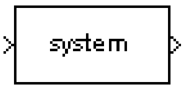
Purpose

Access an S-function.

Library

Nonlinear

Description



The S-Function block provides access to S-functions from a block diagram. The S-function named as the **S-function name** parameter can be an M-file or MEX-file written as an S-function.

The S-Function block allows additional parameters to be passed directly to the named S-function. The function parameters can be specified as MATLAB expressions or as variables separated by commas. For example:

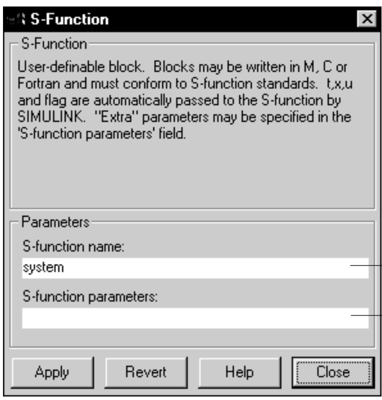
```
A, B, C, D, [eye(2,2);zeros(2,2)]
```

Note that although individual parameters can be enclosed in square brackets, the list of parameters must not be enclosed in square brackets.

The S-function block displays the name of the specified S-function and is always drawn with one input port and one output port, regardless of the number of inputs and outputs of the contained subsystem.

Vector lines are used when the S-function contains more than one input or output. The input vector width must match the number of inputs contained in the S-function. The block directs the first element of the input vector to the first input of the S-function, the second element to the second input, and so on. Likewise, the output vector width must match the number of S-function outputs.

Parameters and Dialog Box



The S-function name.

Additional S-function parameters.

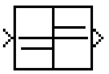
Characteristics

Direct Feedthrough	Depends on contents of S-function
Sample Time	Depends on contents of S-function
Scalar Expansion	Depends on contents of S-function
Vectorized	Depends on contents of S-function
Zero Crossing	No

Purpose Indicate the sign of the input.

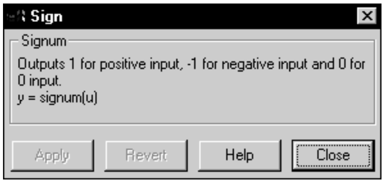
Library Nonlinear

Description The Sign block indicates the sign of the input:



- The output is 1 when the input is greater than zero.
- The output is 0 when the input is equal to zero.
- The output is -1 when the input is less than zero.

Dialog Box



Characteristics

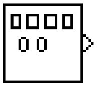
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Vectorized	Yes
Zero Crossing	Yes, to detect when the input crosses through zero

Signal Generator

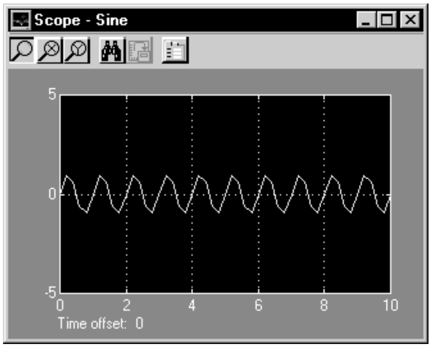
Purpose Generate various waveforms.

Library Sources

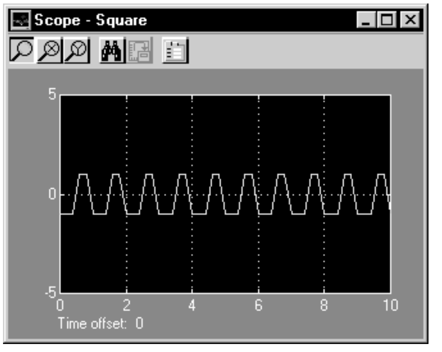
Description



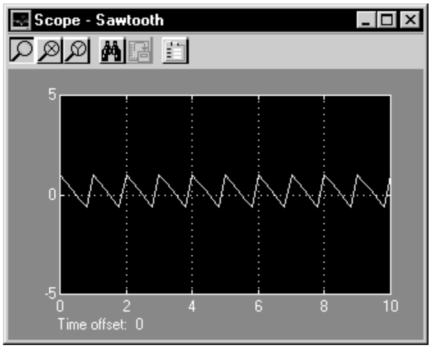
The Signal Generator block can produce one of three different waveforms: sine wave, square wave, and sawtooth wave. The signal parameters can be expressed in Hertz (the default) or radians per second. This figure shows each signal displayed on a Scope using default parameter values:



Sine Wave



Square Wave

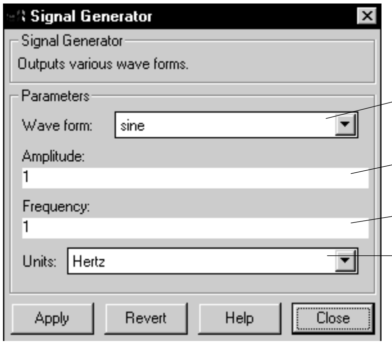


Sawtooth Wave

A negative **Amplitude** parameter value causes a 180-degree phase shift. You can generate a phase-shifted wave at other than 180 degrees in a variety of ways, including inputting a Clock block signal to a MATLAB Fcn block and writing the equation for the particular wave.

You can vary the output settings of the Signal Generator block while a simulation is in progress. This is useful to determine quickly the response of a system to different types of inputs.

Parameters and Dialog Box



The wave form: a sine wave, square wave, or sawtooth wave. The default is a sine wave.

The signal amplitude. The default is 1.

The signal frequency. The default is 1.

The signal units, Hertz or radians/sec. The default is Hertz.

Characteristics

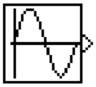
Sample Time	Inherited
Scalar Expansion	Of parameters
Vectorized	Yes
Zero Crossing	No

Sine Wave

Purpose Generate a sine wave.

Library Sources

Description



The Sine Wave block provides a sinusoid. The block can operate in either continuous or discrete mode.

The output of the Sine Wave block is determined by:

$$y = Amplitude \times \sin(frequency \times time + phase)$$

The value of the **Sample time** parameter determines whether the block operates in continuous mode or discrete mode:

- `0` (the default) causes the block to operate in continuous mode.
- `>0` causes the block to operate in discrete mode.
- `-1` causes the block to operate in the same mode as the block receiving the signal.

Using the Sine Wave Block in Discrete Mode

A **Sample time** parameter value greater than zero causes the block to behave as if it were driving a Zero-Order Hold block whose sample time is set to that value.

Using the Sine Wave block in this way allows you to build models with sine wave sources that are purely discrete, rather than models that are hybrid continuous/discrete systems. Hybrid systems are inherently more complex and, as a result, take longer to simulate.

The Sine Wave block in discrete mode uses an incremental algorithm rather than one based on absolute time. As a result, the block can be useful in models intended to run for an indefinite length of time, such as in vibration or fatigue testing.

The incremental algorithm computes the sine based on the value computed at the previous sample time. This method makes use of the following identities:

$$\sin(t + \Delta t) = \sin(t)\cos(\Delta t) + \sin(\Delta t)\cos(t)$$
$$\cos(t + \Delta t) = \cos(t)\cos(\Delta t) - \sin(t)\sin(\Delta t)$$

These identities can be written in matrix form:

$$\begin{bmatrix} \sin(t+\Delta t) \\ \cos(t+\Delta t) \end{bmatrix} = \begin{bmatrix} \cos(\Delta t) & \sin(\Delta t) \\ -\sin(\Delta t) & \cos(\Delta t) \end{bmatrix} \begin{bmatrix} \sin(t) \\ \cos(t) \end{bmatrix}$$

Since Δt is constant, the following expression is a constant.

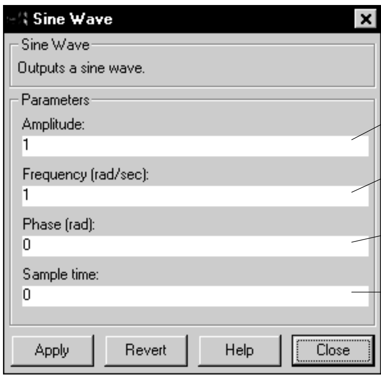
$$\begin{bmatrix} \cos(\Delta t) & \sin(\Delta t) \\ -\sin(\Delta t) & \cos(\Delta t) \end{bmatrix}$$

Therefore the problem becomes one of a matrix multiply of the value of sin(t) by a constant matrix to obtain sin($t+\Delta t$). This algorithm may also be faster on computers that do not have hardware floating-point support for trigonometric functions.

Using the Sine Wave Block in Continuous Mode

A **Sample time** parameter value of zero causes the block to behave in continuous mode. When operating in continuous mode, the Sine Wave block can become inaccurate due to loss of precision as time becomes very large.

Parameters and Dialog Box



The amplitude of the signal. The default is 1.

The frequency, in radians/second. The default is 1 rad/sec.

The phase shift, in radians. The default is 0 radians.

The sample period. The default is 0.

Characteristics

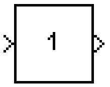
Sample Time	Continuous, discrete, or inherited
Scalar Expansion	Of parameters
Vectorized	Yes
Zero Crossing	No

Slider Gain

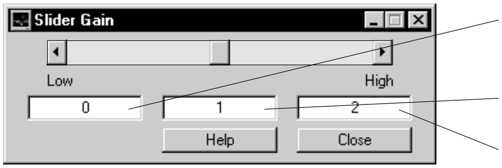
Purpose Vary a scalar gain using a slider.

Library Linear

Description The Slider Gain block allows you to vary a scalar gain during a simulation using a slider. The block accepts one input and generates one output.



Dialog Box



The edit fields indicate (from left to right) the lower limit, the current value, and the upper limit. You can change the gain in two ways: by manipulating the slider, or by entering a new value in the current value field. You can change the range of gain values by changing the lower and upper limits. Close the dialog box by clicking on the **Close** button.

If you click on the slider's left or right arrow, the current value changes by about 1% of the slider's range. If you click on the rectangular area to either side of the slider's indicator, the current value changes by about 10% of the slider's range.

To apply a vector gain to the block input, consider using the Gain block, described on page 9–70. To apply a matrix gain, use the Matrix Gain block, described on page 9–96.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of the gain
States	0
Vectorized	Yes
Zero Crossing	No

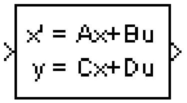
Purpose

Implement a linear state-space system.

Library

Linear

Description

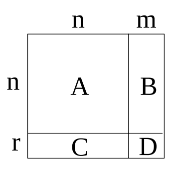


The State-Space block implements a system whose behavior is defined by:

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

where x is the state vector, u is the input vector, and y is the output vector. The matrix coefficients must have these characteristics, as illustrated in the diagram below:

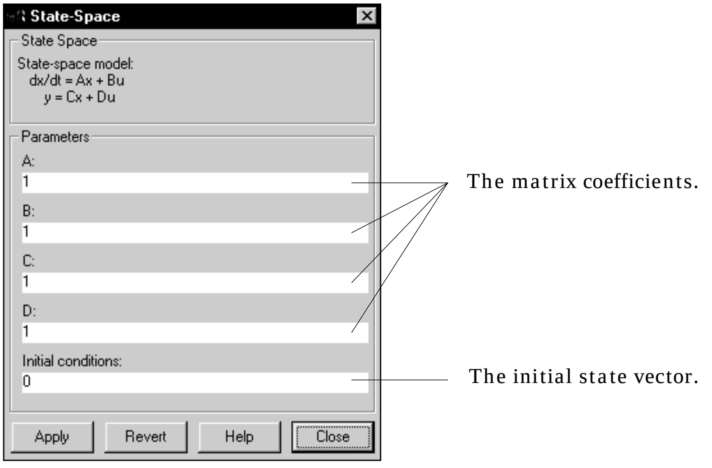
- **A** must be an n-by-n matrix, where n is the number of states.
- **B** must be an n-by-m matrix, where m is the number of inputs.
- **C** must be an r-by-n matrix, where r is the number of outputs.
- **D** must be an r-by-m matrix.



The block accepts one input and generates one output. The input vector width is determined by the number of columns in the B and D matrices. The output vector width is determined by the number of rows in the C and D matrices.

Simulink converts a matrix containing zeros to a sparse matrix for efficient multiplication.

Parameters and Dialog Box



Characteristics

Direct Feedthrough	Only if $D \neq 0$
Sample Time	Continuous
Scalar Expansion	Of the initial conditions
States	Depends on the size of A
Vectorized	Yes
Zero Crossing	No

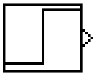
Purpose

Generate a step function.

Library

Sources

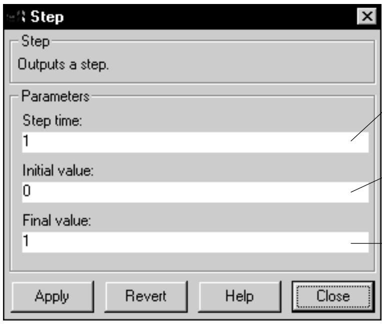
Description



The Step block provides a step between two definable levels at a specified time. If the simulation time is less than the **Step time** parameter value, the block's output is the **Initial value** parameter value. For simulation time greater than or equal to the **Step time**, the output is the **Final value** parameter value.

The Step block generates a scalar or vector output, depending on the length of the parameters.

Parameters and Dialog Box



The time, in seconds, when the output jumps from the **Initial value** parameter to the **Final value** parameter. The default is one second.

The block output until the simulation time reaches the **Step time** parameter. The default is 0.

The block output when the simulation time reaches and exceeds the **Step time** parameter. The default is 1.

Characteristics

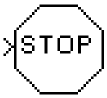
Sample Time	Inherited from driven block
Scalar Expansion	Of parameters
Vectorized	Yes
Zero Crossing	Yes, to detect step times

Stop Simulation

Purpose Stop the simulation when the input is nonzero.

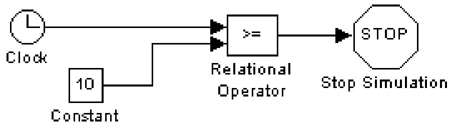
Library Sinks

Description The Stop Simulation block stops the simulation when the input is nonzero.

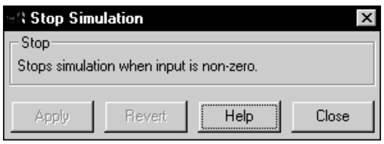


The simulation completes the current time step before terminating. If the block input is a vector, any nonzero vector element causes the simulation to stop.

You can use this block in conjunction with the Relational Operator block to control when the simulation stops. For example, this model stops the simulation when the input signal reaches 10:



Dialog Box



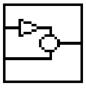
Characteristics

Sample Time	Inherited from driving block
Vectorized	Yes

Purpose Represent a system within another system.

Library Connections

Description A Subsystem block represents a system within another system. You create a subsystem in these ways:



- Copy the Subsystem block from the Connections library into your model. You can then add blocks to the subsystem by opening the Subsystem block and copying blocks into its window.
- Select the blocks and lines that are to make up the subsystem using a bounding box, then choose **Create Subsystem** from the **Edit** menu. Simulink replaces the blocks with a Subsystem block. When you open the block, the window displays the blocks you selected, adding Inport and Outport blocks to reflect signals entering and leaving the subsystem.

The number of input ports drawn on the Subsystem block's icon corresponds to the number of Inport blocks in the subsystem. Similarly, the number of output ports drawn on the block corresponds to the number of Outport blocks in the subsystem. If Inport and Outport block names are not hidden, they appear as port labels on the Subsystem block.

For more information about subsystems, see "Creating Subsystems" in Chapter 3.

Dialog Box None

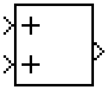
Characteristics

Sample Time	Depends on the blocks in the subsystem
Vectorized	Depends on the blocks in the subsystem
Zero Crossing	Yes, for enable and trigger ports if present

Purpose Generate the sum of inputs.

Library Linear

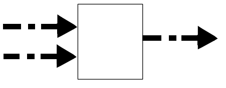
Description The Sum block adds scalar and/or vector inputs, or elements of a single vector input, depending on the number of block inputs:



- If the block has more than one input, the block output is an element-by-element sum across the inputs. If all inputs are scalars, the output is a scalar. For a block having n inputs, if any input is a vector, each element of the output is generated as:

  $$y_i = u1_i + u2_i + \ldots + un_i$$

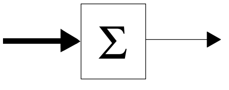
  This model represents using the Sum block in this way. The broken lines indicate that each signal can be a scalar or vector. The output is a scalar only if all inputs are scalars.

- If the block has one vector input, the block output is the scalar sum of the elements of the input:

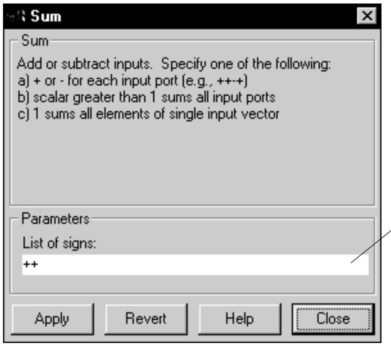
  $$y = \Sigma u_i$$

  This model represents using the Sum block in this way. The solid line input signal indicates that the input is a vector.

The Sum block draws plus and minus signs beside the appropriate ports and redraws its ports to match the number of signs specified in the **List of signs** parameter. If the number of signs is changed, ports are added or deleted from the bottom of the icon. If necessary, Simulink resizes the block to show all input ports.

Parameters and Dialog Box



A constant or a combination of + and – symbols. Specifying a constant causes Simulink to redraw the block with that number of ports, all with positive polarity. A combination of plus and minus signs specifies the polarity of each port, where the number of ports equals the number of symbols used.

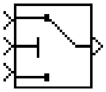
Characteristics

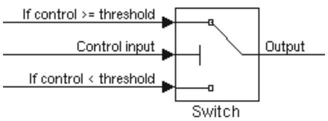
Direct Feedthrough	Yes
Sample Time	Inherited from driving blocks
Scalar Expansion	Yes
States	0
Vectorized	Yes
Zero Crossing	No

Purpose Switch between two inputs.

Library Nonlinear

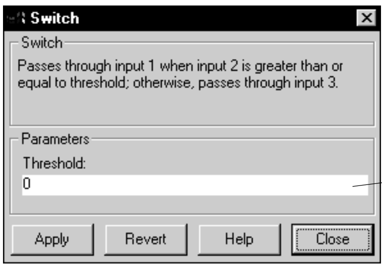
Description The Switch block propagates one of two inputs to its output depending on the value of a third input, called the control input. If the signal on the control (second) input is greater than or equal to the **Threshold** parameter, the block propagates the first input; otherwise, it propagates the third input. This figure shows the use of the block ports:





To drive the switch with a logic input (i.e., 0 or 1), set the threshold to 0.5.

Parameters and Dialog Box



The value of the control (the second input) at which the switch flips to its other state. You can specify this parameter as either a scalar or a vector equal in width to the input vectors.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Vectorized	Yes
Zero Crossing	Yes, to detect when the switch condition occurs

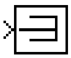
Purpose

Terminate an unconnected output port.

Library

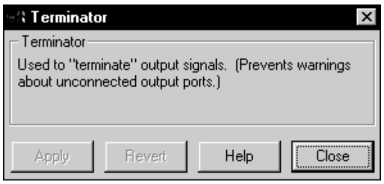
Connections

Description



The Terminator block can be used to cap blocks whose output ports are not connected to other blocks. If you run a simulation with blocks having unconnected output ports, Simulink issues warning messages. Using Terminator blocks to cap those blocks avoids warning messages.

Parameters and Dialog Box



Characteristics

Sample Time	Inherited from driving block
Vectorized	Yes

To File

Purpose

Write data to a file.

Library

Sinks

Description

The To File block writes its input to a matrix in a MAT-file. The block writes one column for each time step: the first row is the simulation time; the remainder of the column is the input data, one data point for each element in the input vector. The matrix has this form:

$$\begin{bmatrix} t_1 & t_2 & \ldots t_{final} \\ u1_1 & u1_2 & \ldots u1_{final} \\ .. & & \\ un_1 & un_2 & \ldots un_{final} \end{bmatrix}$$

The From File block can use data written by a To File block without any modifications. However, the form of the matrix expected by the From Workspace block is the transpose of the data written by the To File block.

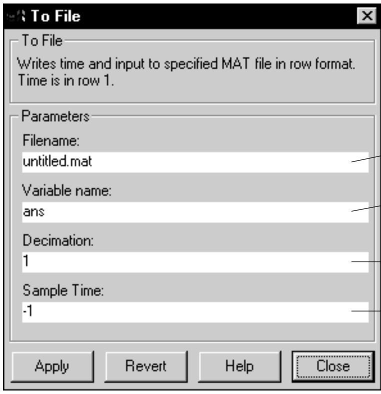
The block writes the data as well as the simulation time after the simulation is completed. The block icon shows the name of the specified output file.

The amount of data written and the time steps at which the data is written are determined by block parameters:

- The **Decimation** parameter allows you to write data at every nth sample, where n is the decimation factor. The default decimation, 1, writes data at every time step.
- The **Sample time** parameter allows you to specify a sampling interval at which to collect points. This parameter is useful when using a variable-step solver where the interval between time steps may not be the same. The default value of -1 causes the block to inherit the sample time from the driving block when determining which points to write.

If the file exists at the time the simulation starts, the block overwrites its contents.

Parameters and Dialog Box



The name of the MAT-file that holds the matrix.

The name of the matrix contained in the named file.

A decimation factor. The default value is 1.

The sample time at which to collect points.

Characteristics

Sample Time	Inherited from driving block
Vectorized	Yes

To Workspace

Purpose Write data to a matrix in the workspace.

Library Sinks

Description

The To Workspace block writes its input to the specified matrix in the workspace. The block writes its input row by row, where each row consists of all input vector elements at a time step. If the matrix already exists, its contents are overwritten. The matrix has this form:

$$\begin{bmatrix} u1_1 & u2_1 & \ldots & un_1 \\ u1_2 & u2_2 & \ldots & un_2 \\ \ldots & & & \\ u1_{final} & u2_{final} & \ldots & un_{final} \end{bmatrix}$$

The amount of data written and the time steps at which the data is written are determined by block parameters:

- The **Maximum number of rows** parameter indicates how many data rows to save. If the simulation generates more rows than the specified maximum, the simulation saves only the most recently generated rows. To capture all the data, set this value to `inf`.
- The **Decimation** parameter allows you to write data at every `n`th sample, where `n` is the decimation factor. The default decimation, 1, writes data at every time step.
- The **Sample time** parameter allows you to specify a sampling interval at which to collect points. This parameter is useful when using a variable-step solver where the interval between time steps may not be the same. The default value of -1 causes the block to inherit the sample time from the driving block when determining which points to write.

During the simulation, the block writes data to an internal buffer. When the simulation is completed or paused, that data is written to the workspace. The block icon shows the name of the matrix to which the data is written.

Using Saved Data with a From File Block

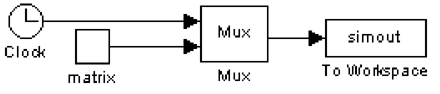
If the data written using a To Workspace block is to be saved and read later by a From File block, the time must be added to the data and the matrix must be

transposed. For more information, see the description of the From File block on page 9–66.

Using Saved Data with a From Workspace Block

If the data written using a To Workspace block is intended to be “played back” in another simulation using a From Workspace block, the first column of the data must contain the simulation time values. You can add a column with time values in two ways:

- By multiplexing the output of a Clock block as the first element of the vector input line of the To Workspace block.



- By specifying time as a return value on the **Simulation Parameters** dialog box or from the command line, described in Chapter 4. When the simulation is completed, you can concatenate the time vector (`t`) to the matrix using a command like this:

```
matrix = [t; matrix];
```

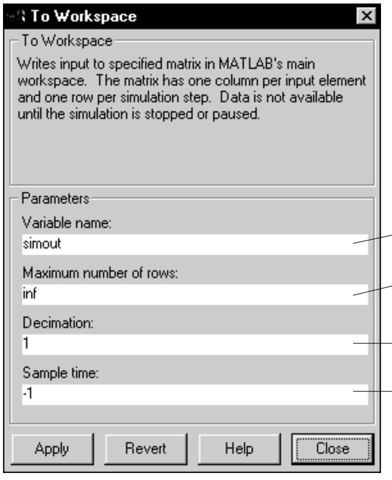
Examples

In a simulation where the start time is 0, the **Maximum number of rows** is 100, the **Decimation** is 1, and the **Sample time** is 0.5. The To Workspace block collects a maximum of 100 points, at time values of 0, 0.5, 1.0, 1.5, ... seconds. Specifying a **Decimation** of 1 directs the block to write data at each step.

In a similar example, the **Maximum number of rows** is 100 and the **Sample time** is 0.5, but the **Decimation** is 5. In this example, the block collects up to 100 points, at time values of 0, 2.5, 5.0, 7.5, ... seconds. Specifying a **Decimation** of 5 directs the block to write data at every fifth sample. The sample time ensures that data is written at these points.

In another example, all parameters are as defined in the first example except that the **Maximum number of rows** is 3. In this case, only the last three rows collected are written to the workspace. If the simulation stop time is 100, data corresponds to times 99.0, 99.5, and 100.0 seconds (three points).

Parameters and Dialog Box



The name of the matrix that holds the data.

The maximum number of rows (one row per time step) to be saved. The default is 1000 rows.

A decimation factor. The default is 1.

The sample time at which to collect points.

Characteristics

Sample Time	Inherited
Vectorized	Yes

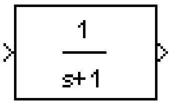
Purpose

Implement a linear transfer function.

Library

Linear

Description



The Transfer Fcn block implements a transfer function where the input (u) and output (y) can be expressed in transfer function form as the following equation:

$$H(s) = \frac{y(s)}{u(s)} = \frac{num(s)}{den(s)} = \frac{num(1)s^{nn-1} + num(2)s^{nn-2} + \ldots + num(nn)}{den(1)s^{nd-1} + den(2)s^{nd-2} + \ldots + den(nd)}$$

where nn and nd are the number of numerator and denominator coefficients, respectively. num and den contain the coefficients of the numerator and denominator in descending powers of s. num can be a vector or matrix, den must be a vector, and both are specified as parameters on the block dialog box. The order of the denominator must be greater than or equal to the order of the numerator.

Block input is scalar; output width is equal to the number of rows in the numerator.

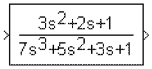
Initial conditions are preset to zero. If you need to specify initial conditions, convert to state-space form using `tf2ss` and use the State-Space block. The `tf2ss` utility provides the A, B, C, and D matrices for the system. For more information, type `help tf2ss` or consult the *Control System Toolbox User's Guide*.

The Transfer Fcn Block Icon

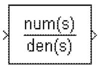
The numerator and denominator are displayed on the Transfer Fcn block icon depending on how they are specified:

- If each is specified as an expression, a vector, or a variable enclosed in parentheses, the icon shows the transfer function with the specified coefficients and powers of s. If you specify a variable in parentheses, the variable is evaluated. For example, if you specify **Numerator** as `[3,2,1]` and

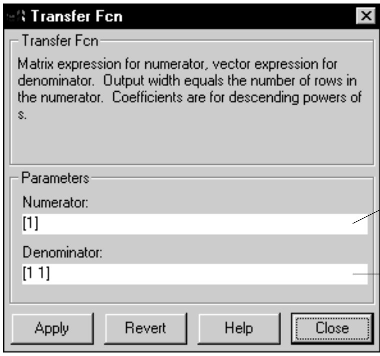
Denominator as (den) where den is [7,5,3,1], the block icon looks like this:



- If each is specified as a variable, the icon shows the variable name followed by "(s)". For example, if you specify **Numerator** as num and **Denominator** as den, the block icon looks like this:



Parameters and Dialog Box



The row vector of numerator coefficients. A matrix with multiple rows can be specified to generate multiple output. The default is [1].

The row vector of denominator coefficients. The default is [1 1].

Characteristics

Direct Feedthrough	Only if the lengths of the **Numerator** and **Denominator** parameters are equal
Sample Time	Continuous
Scalar Expansion	No
States	Length of **Denominator** -1
Vectorized	Yes
Zero Crossing	No

Purpose

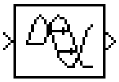
Delay the input by a given amount of time.

Library

Nonlinear

Description

The Transport Delay block delays the input by a specified amount of time. It can be used to simulate a time delay.



At the start of the simulation, the block outputs the **Initial input** parameter until the simulation time exceeds the **Time delay** parameter, when the block begins generating the delayed input. The **Time delay** parameter must be nonnegative.

The block stores input points and simulation times during a simulation in a buffer whose initial size is defined by the **Initial buffer size** parameter. If the number of points exceeds the buffer size, the block allocates additional memory and Simulink displays a message after the simulation that indicates the total buffer size needed. Because allocating memory slows down the simulation, define this parameter value carefully if simulation speed is an issue. For long time delays, this block might use a large amount of memory, particularly for a vectorized input.

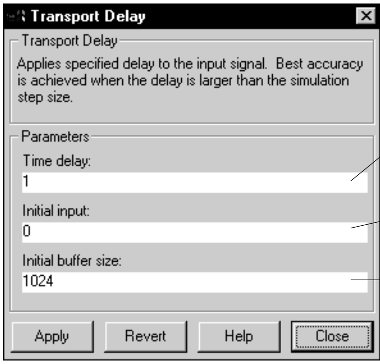
When output is required at a time that does not correspond to the times of the stored input values, the block interpolates linearly between points. When the delay is smaller than the step size, the block extrapolates from the last output point, which may produce inaccurate results. Because the block does not have direct feedthrough, it cannot use the current input to calculate its output value. To illustrate this point, consider a fixed-step simulation with a step size of 1 and the current time at t = 5. If the delay is 0.5, the block needs to generate a point at t = 4.5. Because the most recent stored time value is at t = 4, the block performs forward extrapolation.

The Transport Delay block does not interpolate discrete signals. Instead, it returns the discrete value at *t - tdelay*.

This block differs from the Unit Delay block, which delays and holds the output on sample hits only.

Using `linmod` to linearize a model that contains a Transport Delay block can be troublesome. For more information about ways to avoid the problem, see “Linearization” in Chapter 5.

Parameters and Dialog Box



The amount of simulation time that the input signal is delayed before propagating it to the output. The value must be nonnegative.

The output generated by the block between the start of the simulation and the **Time delay**.

The initial memory allocation for the number of points to store.

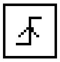
Characteristics

Direct Feedthrough	No
Sample Time	Continuous
Scalar Expansion	Of input and all parameters except **Initial buffer size**
Vectorized	Yes
Zero Crossing	No

Purpose Add a trigger port to a subsystem.

Library Connections

Description



Adding a Trigger block to a subsystem makes it a triggered subsystem. A triggered subsystem executes once on each integration step when the value of the signal that passes through the trigger port changes in a specifiable way (described below). A subsystem can contain no more than one Trigger block. For more information about triggered subsystems, see Chapter 7.

The **Trigger type** parameter allows you to choose the type of event that triggers execution of the subsystem:

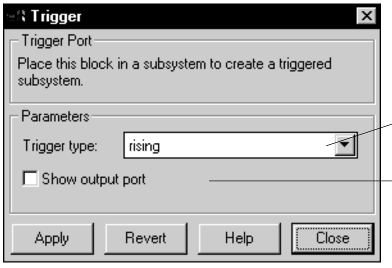
- **rising** triggers execution of the subsystem when the control signal rises from zero to a positive value.
- **falling** triggers execution of the subsystem when the control signal falls from zero to a negative value.
- **either** triggers execution of the subsystem when the control signal either rises from zero to a positive value or falls from zero to a negative value.
- **function-call** causes execution of the subsystem to be controlled by logic internal to an S-function (for more information, see “Function-Call Subsystems” in Chapter 7).

You can output the trigger signal by selecting the **Show output port** check box. Selecting this option allows the system to determine what caused the trigger. The width of the signal is the width of the triggering signal. The signal value is:

- 1 for a signal that causes a rising trigger
- -1 for a signal that causes a falling trigger
- 0 otherwise

Trigger

Parameters and Dialog Box



The type of event that triggers execution of the subsystem

If checked, Simulink draws the Trigger block output port and outputs the trigger signal.

Characteristics

Sample Time	Determined by the signal at the trigger port
Vectorized	Yes

Purpose

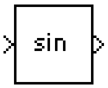
Perform a trigonometric function.

Library

Nonlinear

Description

The Trigonometric Function block performs numerous common trigonometric functions.

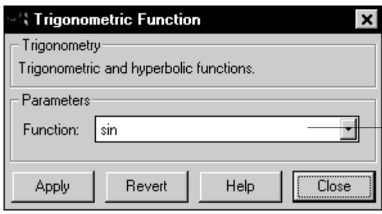


You can select one of these functions from the **Function** list: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `sinh`, `cosh`, and `tanh`. The block output is the result of the function operating on the input or inputs.

The name of the function appears on the block icon. Simulink automatically draws the appropriate number of input ports.

Use the Trigonometric Function block instead of the Fcn block when you want vectorized output because the Fcn block can produce only scalar output.

Parameters and Dialog Box



The trigonometric function.

Characteristics

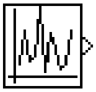
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of the input when the function requires two inputs
Vectorized	Yes
Zero Crossing	No

Uniform Random Number

Purpose Generate uniformly distributed random numbers.

Library Sources

Description

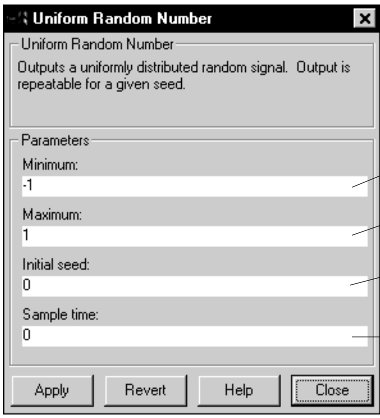


The Uniform Random Number block generates uniformly distributed random numbers over a specifiable interval with a specifiable starting seed. The seed is reset each time a simulation starts. The generated sequence is repeatable and can be produced by any Uniform Random Number block with the same seed and parameters. To generate a vector of random numbers, specify the **Initial seed** parameter as a vector.

To generate normally distributed random numbers, use the Random Number block, described on page 9–111.

Avoid integrating a random signal because solvers are meant to integrate relatively smooth signals. Instead, use the Band-Limited White Noise block.

Parameters and Dialog Box



The minimum of the interval. The default is -1.

The maximum of the interval. The default is 1.

The starting seed for the random number generator. The default is 0.

The sample period. The default is 0.

Characteristics

Sample Time	Continuous, discrete, or inherited
Scalar Expansion	No
Vectorized	Yes
Zero Crossing	No

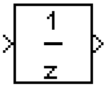
Purpose

Delay a signal one sample period.

Library

Discrete

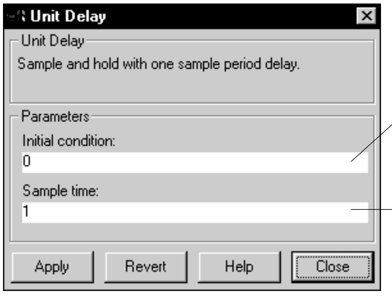
Description



The Unit Delay block delays and holds its input signal by one sampling interval. If the input to the block is a vector, all elements of the vector are delayed by the same sample delay. This block is equivalent to the z^{-1} discrete-time operator.

If an undelayed sample-and-hold function is desired, use a Zero-Order Hold block, or if a delay of greater than one unit is desired, use a Discrete Transfer Fcn block. (See the description of the Transport Delay block for an example that uses the Unit Delay block.)

Parameters and Dialog Box



The block output for the first simulation period, during which the output of the Unit Delay block is undefined. Careful selection of this parameter can minimize unwanted output behavior during this time. The default is 0.

The time interval between samples. The default is 1.

Characteristics

Direct Feedthrough	No
Sample Time	Discrete
Scalar Expansion	Of the **Initial condition** parameter or the input
States	Inherited from driving block or parameters
Vectorized	Yes
Zero Crossing	No

Variable Transport Delay

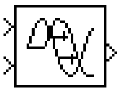
Purpose

Delay the input by a variable amount of time.

Library

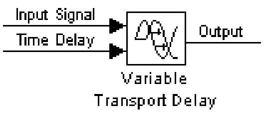
Nonlinear

Description



The Variable Transport Delay block can be used to simulate a variable time delay. The block might be used to model a system with a pipe where the speed of a motor pumping fluid in the pipe is variable.

The block accepts two inputs: the first input is the signal that passes through the block; the second input is the time delay, as show in this icon:



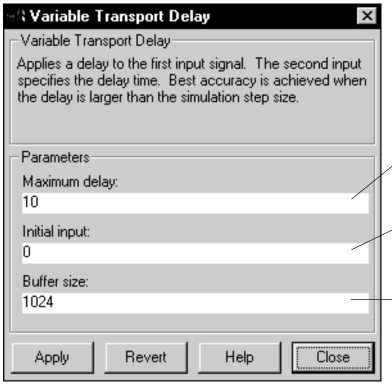
The **Maximum delay** parameter defines the largest value the time delay input can have. The block clips values of the delay that exceed this value. The **Maximum delay** must be greater than or equal to zero. If the time delay becomes negative, the block clips it to zero and issues a warning message.

During the simulation, the block stores time and input value pairs in an internal buffer. At the start of the simulation, the block outputs the **Initial input** parameter until the simulation time exceeds the time delay input. Then, at each simulation step the block outputs the signal at the time that corresponds to the current simulation time minus the delay time.

When output is required at a time that does not correspond to the times of the stored input values, the block interpolates linearly between points. If the time delay is smaller than the step size, the block extrapolates an output point. This may result in less accurate results. The block cannot use the current input to calculate its output value because the block does not have direct feedthrough at this port. To illustrate this point, consider a fixed-step simulation with a step size of 1 and the current time at t = 5. If the delay is 0.5, the block needs to generate a point at t = 4.5. Because the most recent stored time value is at t = 4, the block performs forward extrapolation.

The Variable Transport Delay block does not interpolate discrete signals. Instead, it returns the discrete value at *t - tdelay*.

Parameters and Dialog Box



The maximum value of the time delay input. The value cannot be negative. The default is 10.

The output generated by the block until the simulation time first exceeds the time delay input. The default is 0.

The number of points the block can store. The default is 1024.

Characteristics

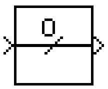
Direct Feedthrough	Yes, of the time delay (second) input
Sample Time	Continuous
Scalar Expansion	Of input and all parameters except **Buffer size**
Vectorized	Yes
Zero Crossing	No

Width

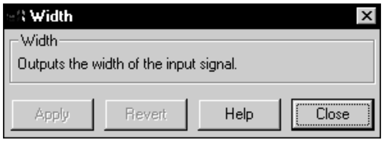
Purpose Output the width of the input vector.

Library Connections

Description The Width block generates as output the width of its input vector.



Parameters and Dialog Box



Characteristics

Sample Time	Constant
Vectorized	Yes

XY Graph

Purpose

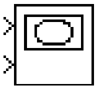
Display an X-Y plot of signals using a MATLAB figure window.

Library

Sinks

Description

The XY Graph block displays an X-Y plot of its inputs in a MATLAB figure window.

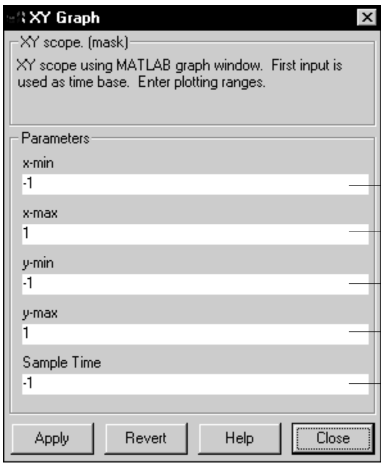


The block has two scalar inputs. The block plots data in the first input (the *x* direction) against data in the second input (the *y* direction). This block is useful for examining limit cycles and other two-state data. Data outside the specified range is not displayed.

Simulink opens a figure window for each XY Graph block in the model at the start of the simulation.

For a demo that illustrates the use of the XY Graph block, enter `lorenzs` in the command window.

Parameters and Dialog Box



The minimum *x*-axis value. The default is -1.

The maximum *x*-axis value. The default is 1.

The minimum *y*-axis value. The default is -1.

The maximum *y*-axis value. The default is 1.

The time interval between samples. The default is -1, which means that the sample time is determined by the driving block.

Characteristics

Sample Time	Inherited from driving block
States	0

Zero-Order Hold

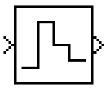
Purpose

Implement zero-order hold of one sample period.

Library

Discrete

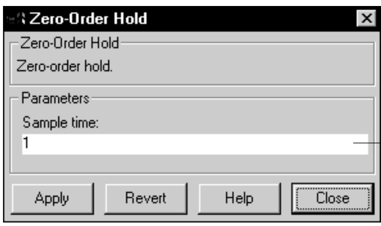
Description



The Zero-Order Hold block implements a sample-and-hold function operating at the specified sampling rate. The block accepts one input and generates one output, both of which can be scalar or vector.

This block provides a mechanism for discretizing one or more signals or resampling the signal at a different rate. You can use it in instances where you need to model sampling without requiring one of the other more complex discrete function blocks. For example, it could be used in conjunction with a Quantizer block to model an A/D converter with an input amplifier.

Parameters and Dialog Box



The time interval between samples. The default is 1.

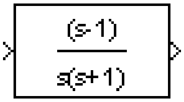
Characteristics

Direct Feedthrough	Yes
Sample Time	Discrete
Scalar Expansion	Yes
States	0
Vectorized	Yes
Zero Crossing	No

Purpose Implement a transfer function specified in terms of poles and zeros.

Library Linear

Description The Zero-Pole block implements a system with the specified zeros, poles, and gain in terms of the Laplace operator s.



A transfer function can be expressed in factored or zero-pole-gain form, which, for a single-input single-output system in MATLAB, is:

$$H(s) = K\frac{Z(s)}{P(x)} = K\frac{(s-Z(1))(s-Z(2))\ldots(s-Z(m))}{(s-P(1))(s-P(2))\ldots(s-P(n))}$$

where Z represents the zeros vector, P the poles vector, and K the gain. Z can be a vector or matrix, P must be a vector, K can be a scalar or vector whose length equals the number of rows in Z. The number of poles must be greater than or equal to the number of zeros. If the poles and zeros are complex, they must be complex conjugate pairs.

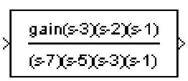
Block input and output widths are equal to the number of rows in the zeros matrix.

The Zero-Pole Block Icon

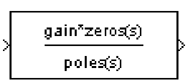
The Zero-Pole block displays the transfer function in its icon depending on how the parameters are specified:

- If each is specified as an expression or a vector, the icon shows the transfer function with the specified zeros, poles, and gain. If you specify a variable in parentheses, the variable is evaluated.

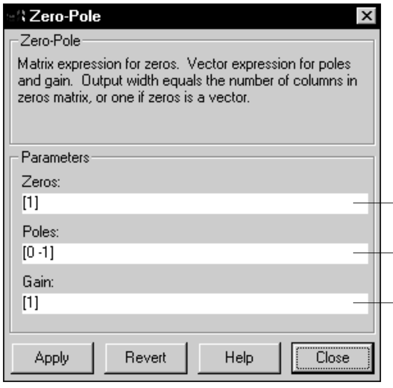
  For example, if you specify **Zeros** as `[3,2,1]`, **Poles** as `(poles)`, where `poles` is defined in the workspace as `[7,5,3,1]`, and **Gain** as `gain`, the icon looks like this:

- If each is specified as a variable, the icon shows the variable name followed by "`(s)`" if appropriate. For example, if you specify **Zeros** as `zeros`, **Poles** as `poles`, and **Gain** as `gain`, the icon looks like this:

Parameters and Dialog Box



The matrix of zeros. The default is `[1]`.

The vector of poles. The default is `[0 -1]`.

The vector of gains. The default is `[1]`.

Characteristics

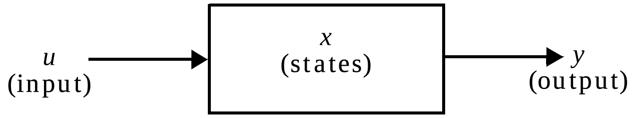
Direct Feedthrough	Only if the lengths of the **Poles** and **Zeros** parameters are equal
Sample Time	Continuous
Scalar Expansion	No
States	Length of **Poles** vector
Vectorized	No
Zero Crossing	No

10

Additional Topics

How Simulink Works

Each block within a Simulink model has these general characteristics: a vector of inputs, u, a vector of outputs, y, and a vector of states, x:



The state vector may consist of continuous states, discrete states, or a combination of both. The mathematical relationships between these quantities are expressed by these equations:

$$
\begin{aligned}
y &= f_o(t, x, u) && \text{output} \\
x_{d_{k+1}} &= f_u(t, x, u) && \text{update} \\
x'_c &= f_d(t, x, u) && \text{derivative}
\end{aligned}
$$

$$\text{where} \quad x = \begin{bmatrix} x_c \\ x_{d_k} \end{bmatrix}$$

Simulation consists of two phases: initialization and simulation. During the initialization phase:

1. The block parameters are passed to MATLAB for evaluation. The resulting numerical values are used as the actual block parameters.
2. The model hierarchy is flattened. Each subsystem that is not a conditionally executed subsystem is replaced by the blocks it contains.
3. Blocks are sorted into the order in which they need to be updated. The sorting algorithm constructs a list such that any block with direct feedthrough is not updated until the blocks driving its inputs are updated. It is during this step that algebraic loops are detected. For more information about algebraic loops, see page 10–7.
4. The connections between blocks are checked to ensure that the vector length of the output of each block is the same as the input expected by the blocks it drives.

Now the simulation is ready to run. A model is simulated using numerical integration. Each of the supplied ODE solvers (simulation methods) depends on the ability of the model to provide the derivatives of its continuous states.

Calculating these derivatives is a two-step process. First, each block's output is calculated in the order determined during the sorting. Then, in a second pass, each block calculates its derivatives based on the current time, its inputs, and its states. The resulting derivative vector is returned to the solver, which uses it to compute a new state vector at the next time point. Once a new state vector is calculated, the sampled data blocks and Scope blocks are updated.

Zero Crossings

Simulink uses zero crossings to detect discontinuities in continuous signals. Zero crossings play an important role in:

- The handling of state events
- The accurate integration of discontinuous signals

State Event Handling

A system experiences a *state event* when a change in the value of a state causes the system to undergo a distinct change. A simple example of a state event is a bouncing ball hitting the floor. When simulating such a system using a variable-step solver, the solver typically does not take steps that exactly correspond to the times that the ball makes contact with the floor. As a result, the ball is likely to overshoot the contact point, which results in the ball penetrating the floor.

Simulink uses zero crossings to ensure that time steps occur exactly (within machine precision) at the time state events occur. Because time steps occur at the exact time of contact, the simulation produces no overshoot and the transition from negative to positive velocity is extremely sharp (that is, there is no rounding of corners at the discontinuity). To see a bouncing ball demo, type bounce at the MATLAB prompt.

Integration of Discontinuous Signals

Numerical integration routines are formulated on the assumption that the signals they are integrating are continuous and have continuous derivatives. If a discontinuity (state event) is encountered during an integration step, Simulink uses zero crossing detection to find the time at which the discontinuity occurs. An integration step is then taken up to the left edge of the discontinuity. Finally, Simulink steps over the discontinuity and begins a new integration step on the next piece-wise continuous portion of the signal.

Implementation Details

An example of a Simulink block that uses zero crossings is the Saturation block. Zero crossings detect these state events in the Saturation block:

- The input signal reaches the upper limit
- The input signal leaves the upper limit
- The input signal reaches the lower limit
- The input signal leaves the lower limit

Simulink blocks that define their own state events are considered to have *intrinsic zero crossings*. If you need explicit notification of a zero crossing event, use the Hit Crossing block. See page 10–5 for a list of blocks that incorporate zero crossings.

The detection of a state event depends on the construction of an internal zero crossing signal. This signal is not accessible by the block diagram. For the Saturation block, the signal that is used to detect zero crossings for the upper limit is `zcSignal = UpperLimit – u`, where `u` is the input signal.

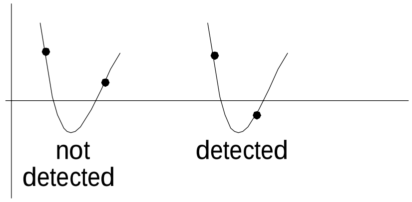
Zero crossing signals have a direction attribute, which can have these values:

- *rising* – a zero crossing occurs when a signal rises to or through zero, or when a signal leaves zero and becomes positive.
- *falling* – a zero crossing occurs when a signal falls to or through zero, or when a signal leaves zero and becomes negative.
- *either* – a zero crossing occurs if either a rising or falling condition occurs.

For the Saturation block's upper limit, the direction of the zero crossing is *either*. This enables the entering and leaving saturation events to be detected using the same zero crossing signal.

If the error tolerances are too large, it is possible for Simulink to fail to detect a zero crossing. For example, if a zero crossing occurs within a time step, but the values at the beginning and end of the step do not indicate a sign change, the solver will step over the crossing without detecting it.

This figure shows a signal that crosses zero. In the first instance, the integrator "steps over" the event. In the second, the solver detects the event.



If you suspect this is happening, tighten the error tolerances to ensure that the solver takes small enough steps. For more information, see "Error Tolerances" on page 4–11.

Caveat

It is possible to create models that exhibit high frequency fluctuations about a discontinuity (chattering). Such systems typically are not physically realizable; a mass-less spring, for example. Because chattering causes repeated detection of zero crossings, the step sizes of the simulation become very small, essentially halting the simulation.

If you suspect that this behavior applies to your model, you can disable zero crossings by selecting the **Disable zero crossing detection** check box on the **Diagnostics** page of the **Simulation Parameters** dialog box. Although disabling zero crossing detection may alleviate the symptoms of this problem, you no longer benefit from the increased accuracy that zero crossing detection provides. A better solution is to try to identify the source of the underlying problem in the model.

These blocks incorporate zero crossings:

Table 10-1: Blocks With Intrinsic Zero Crossings

Block	Description of Zero Crossing
Abs	One: to detect when the input signal crosses zero in either the rising or falling direction.
Backlash	Two: one to detect when the upper threshold is engaged, and one to detect when the lower threshold is engaged.

Table 10-1: Blocks With Intrinsic Zero Crossings (Continued)

Block	Description of Zero Crossing
Dead Zone	Two: one to detect when the dead zone is entered (the input signal minus the lower limit), and one to detect when the dead zone is exited (the input signal minus the upper limit).
Hit Crossing	One: to detect when the input crosses the threshold. These zero crossings are not affected by the **Disable zero crossing detection** check box in the **Simulation Parameters** dialog box.
Integrator	If the reset port is present, to detect when a reset occurs. If the output is limited, there are three zero crossings: one to detect when the upper saturation limit is reached, one to detect when the lower saturation limit is reached, and one to detect when saturation is left.
MinMax	One: for each element of the output vector, to detect when an input signal is the new minimum or maximum
Relay	One: if the relay is off, to detect the switch on point. If the relay is on, to detect the switch off point.
Relational Operator	One: to detect when the output changes.
Saturation	Two: one to detect when the upper limit is reached or left, and one to detect when the lower limit is reached or left.
Sign	One: to detect when the input crosses through zero.
Step	One: to detect the step time.
Subsystem	For conditionally executed subsystems: one for the enable port if present, and one for the trigger port, if present.
Switch	One: to detect when the switch condition occurs.

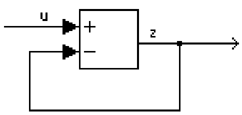
Algebraic Loops

Some Simulink blocks have input ports with *direct feedthrough*. This means that the output of these blocks cannot be computed without knowing the values of the signals entering the blocks at these input ports. Some examples of blocks with direct feedthrough inputs are:

- The Elementary Math block
- The Gain block
- The Integrator block's reset and initial condition ports
- The Product block
- The State-Space block when there is a nonzero D matrix
- The Sum block
- The Transfer Fcn block when the numerator and denominator are of the same order
- The Zero-Pole block when there are as many zeros as poles

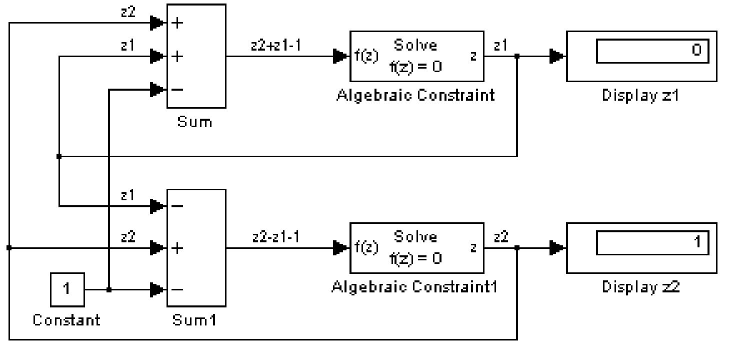
To determine whether a block has direct feedthrough, consult the Characteristics table that describes the block, in Chapter 9.

An *algebraic loop* occurs when an input port with direct feedthrough is driven by the output of the same block, either directly, or by a feedback path through other blocks with direct feedthrough. An example is this simple scalar loop:



Mathematically, this loop implies that the output of the Sum block is an algebraic state z constrained to equal the first input u minus z (i.e. $z = u - z$). The solution of this simple loop is $z = u/2$, but most algebraic loops cannot be

solved by inspection. It is easy to create vector algebraic loops with multiple algebraic state variables *z1*, *z2*, etc., as shown in this model:



The Algebraic Constraint block in the Nonlinear library (described on page 9-10) is a convenient way to model algebraic equations and specify initial guesses. The Algebraic Constraint block constrains its input signal *F(z)* to zero and outputs an algebraic state *z*. This block outputs the value necessary to produce a zero at the input. The output must affect the input through some feedback path. You can provide an initial guess of the algebraic state value in the block's dialog box to improve algebraic loop solver efficiency.

A scalar algebraic loop represents a scalar algebraic equation or constraint of the form $F(z) = 0$, where z is the output of one of the blocks in the loop and the function F consists of the feedback path through the other blocks in the loop to the input of the block. In the simple one-block example shown on the previous page, $F(z) = z - (u - z)$. In the vector loop example shown above, the equations are:

$$z2 + z1 - 1 = 0$$
$$z2 - z1 - 1 = 0$$

Algebraic loops arise when a model includes an algebraic constraint $F(z) = 0$. This constraint may arise as a consequence of the physical interconnectivity of the system you are modeling, or it may arise because you are specifically trying to model a differential/algebraic system (DAE).

When a model contains an algebraic loop, Simulink calls a loop solving routine at each time step. The loop solver performs iterations to determine the solution to the problem (if it can). As a result, models with algebraic loops run slower than models without them.

To solve $F(z) = 0$, the Simulink loop solver uses Newton's method with weak line search and rank-one updates to a Jacobian matrix of partial derivatives. Although the method is robust, it is possible to create loops for which the loop solver will not converge without a good initial guess for the algebraic states z. You can specify an initial guess for a line in an algebraic loop by placing an IC block (which is normally used to specify an initial condition for a signal) on that line. As shown above, another way to specify an initial guess for a line in an algebraic loop is to use an Algebraic Constraint block.

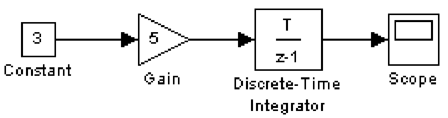
Whenever possible, use an IC block or an Algebraic Constraint block to specify an initial guess for the algebraic state variables in a loop.

Invariant Constants

Blocks either have explicitly defined sample times or inherit their sample times from blocks that feed them or are fed by them.

Simulink assigns Constant blocks a sample time of infinity, also referred to as a *constant sample time*. Other blocks have constant sample time if they receive their input from a Constant block and do not inherit the sample time of another block. This means that the output of these blocks does not change during the simulation unless the parameters are explicitly modified by the model user.

For example, in this model, both the Constant and Gain blocks have constant sample time:



Because Simulink supports the ability to change block parameters during a simulation, all blocks, even blocks having constant sample time, must generate their output at the model’s effective sample time.

Because of this feature, *all* blocks compute their output at each sample time hit, or, in the case of purely continuous systems, at every simulation step. For blocks having constant sample time whose parameters do not change during a simulation, evaluating these blocks during the simulation is inefficient and slows down the simulation.

You can set the `InvariantConstants` parameter to remove all blocks having constant sample times from the simulation “loop.” The effect of this feature is twofold: first, parameters for these blocks cannot be changed during a

simulation; and second, simulation speed is improved. The speed improvement depends on model complexity, the number of blocks with constant sample time, and the effective sampling rate of the simulation.

You can set the parameter for your model by entering this command:

```
set_param('model_name', 'InvariantConstants', 'on')
```

You can turn off the feature by issuing the command again, assigning the parameter the value of `'off'`.

You can determine which blocks have constant sample time by selecting **Sample Time Colors** from the **Format** menu. Blocks having constant sample time are colored magenta.

Discrete-Time Systems

Simulink has the ability to simulate discrete (sampled data) systems. Models can be *multirate*; that is, they can contain blocks that are sampled at different rates. Models can also be *hybrid*, containing a mixture of discrete and continuous blocks.

Discrete Blocks

Each of the discrete blocks has a built-in sampler at its input, and a zero-order hold at its output. When the discrete blocks are mixed with continuous blocks, the output of the discrete blocks between sample times is held constant. The outputs of the discrete blocks are updated only at times that correspond to sample hits.

Sample Time

The **Sample time** parameter sets the sample time at which a discrete block's states are updated. Normally, the sample time is set to a scalar variable; however, it is possible to specify an offset time (or skew) by specifying a two-element vector in this field.

For example, specifying the **Sample time** parameter as the vector `[Ts,offset]` sets the sample time to `Ts` and the offset value to `offset`. The discrete block is updated on integer multiples of the sample time and offset values only:

```
t = n * Ts + offset
```

where `n` is an integer and `offset` can be positive or negative, but less than the sample time. The offset is useful if some discrete blocks must be updated sooner or later than others.

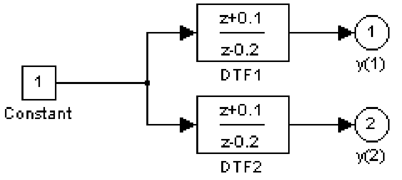
You cannot change the sample time of a block while a simulation is running. If you want to change a block's sample time, you must stop and restart the simulation for the change to take effect.

Purely Discrete Systems

Purely discrete systems can be simulated using any of the solvers; there is no difference in the solutions. To generate output points only at the sample hits, choose one of the `discrete` solvers.

Multirate Systems

Multirate systems contain blocks that are sampled at different rates. These systems can be modeled with discrete blocks or both discrete and continuous blocks. For example, consider this simple multirate discrete model:

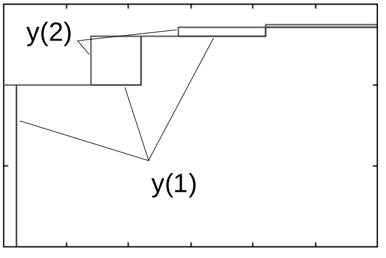


For this example the DTF1 Discrete Transfer Fcn block's **Sample time** is set to `[1 0.1]`, which gives it an offset of `0.1`. The DTF2 Discrete Transfer Fcn block's **Sample time** is set to `0.7`, with no offset.

Starting the simulation and plotting the outputs using the `stairs` function:

```
[t,x,y] = sim('multirate', 3);
stairs(t,y)
```

produces this plot:



For the DTF1 block, which has an offset of `0.1`, there is no output until `t = 0.1`. Because the initial conditions of the transfer functions are zero, the output of DTF1, y(1), is zero before this time.

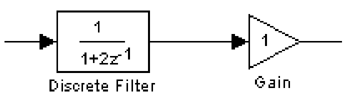
Sample Time Colors

Simulink identifies different sample rates in a model using the sample time color feature, which shows sample rates by applying the color scheme shown in this table:

Table 10-2: Sample Time Colors

Color	Use
Black	Continuous blocks
Magenta	Constant blocks
Yellow	Hybrid (subsystems grouping blocks, or Mux or Demux blocks grouping signals with varying sample times)
Red	Fastest discrete sample time
Green	Second fastest discrete sample time
Blue	Third fastest discrete sample time
Light Blue	Fourth fastest discrete sample time
Dark Green	Fifth fastest discrete sample time
Cyan	Triggered sample time

To understand how this feature works, it is important to be familiar with Simulink's Sample Time Propagation Engine (STPE). The figure below illustrates a Discrete Filter block with a sample time of `Ts` driving a Gain block. Because the Gain block's output is simply the input multiplied by a constant, its output changes at the same rate as the filter. In other words, the Gain block has an effective sample rate equal to that of the filter's sample rate. This is the fundamental mechanism behind the STPE.



To enable the sample time colors feature, select **Sample Time Colors** from the **Format** menu.

Simulink does not automatically recolor the model with each change you make to it, so you must select **Update Diagram** from the **Edit** menu to explicitly update the model coloration. To return to your original coloring, disable sample time coloration by again choosing **Sample Time Colors**.

When using sample time colors, the color assigned to each block depends on its sample time with respect to other sample times in the model.

Simulink sets sample times for individual blocks according to these rules:

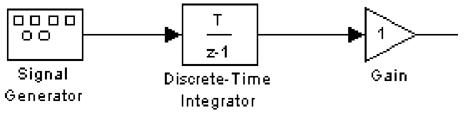
- Continuous blocks (e.g., Integrator, Derivative, Transfer Fcn, etc.) are, by definition, continuous.
- Constant blocks (for example, Constant) are, by definition, constant.
- Discrete blocks (e.g., Zero-Order Hold, Unit Delay, Discrete Transfer Fcn, etc.) have sample times that are explicitly specified by the user on the block dialog boxes.
- All other blocks have implicitly defined sample times that are based on the sample times of their inputs. For instance, a Gain block that follows an Integrator is treated as a continuous block, whereas a Gain block that follows a Zero-Order Hold is treated as a discrete block having the same sample time as the Zero-Order Hold block.

  For blocks whose inputs have different sample times, if all sample times are integer multiples of the fastest sample time, the block is assigned the sample time of the fastest input. If a variable-step solver is being used, the block is assigned the continuous sample time. If a fixed-step solver is being used and the greatest common divisor of the sample times (the fundamental sample time) can be computed, it is used. Otherwise continuous is used.

It is important to note that Mux and Demux blocks are simply grouping operators – signals passing through them retain their timing information. For this reason, the lines emanating from a Demux block may have different colors if they are driven by sources having different sample times. In this case, the Mux and Demux blocks are color coded as hybrids (yellow) to indicate that they handle signals with multiple rates.

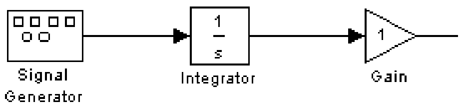
Similarly, Subsystem blocks that contain blocks with differing sample times are also colored as hybrids, because there is no single rate associated with them. If all of the blocks within a subsystem run at a single rate, then the Subsystem block is colored according to that rate.

Under some circumstances, Simulink also backpropagates sample times to source blocks if it can do so without affecting the output of a simulation. For instance, in the model below, Simulink recognizes that the Signal Generator is driving a Discrete-Time Integrator so it assigns the Signal Generator and the Gain block the same sample time as the Discrete-Time Integrator block.



You can verify this by enabling **Sample Time Colors** and noting that all blocks are colored red. Because the Discrete-Time Integrator block only looks at its input at its sample times, this change does not affect the outcome of the simulation but does result in a performance improvement.

Replacing the Discrete-Time Integrator block with a continuous Integrator block, as shown below, and recoloring the model by choosing **Update Diagram** from the **Edit** menu cause the Signal Generator and Gain blocks to change to continuous blocks, as indicated by their being colored black.



Mixed Continuous and Discrete Systems

Mixed continuous and discrete systems are composed of both sampled and continuous blocks. Such systems can be simulated using any of the integration methods, although certain methods are more efficient and accurate than others. For most mixed continuous and discrete systems, the Runge-Kutta variable step methods, `ode23` and `ode45`, are superior to the other methods in terms of efficiency and accuracy. Due to discontinuities associated with the sample and hold of the discrete blocks, the `ode15s` and `ode113` methods are not recommended for mixed continuous and discrete systems.

11

Model Construction Commands

Introduction

This table indicates the tasks performed by the commands described in this chapter. The reference section of this chapter lists the commands in alphabetical order.

Task	Command
Create a new system	`new_system`
Open an existing system	`open_system`
Close a system window	`close_system`, `bdclose`
Save a Simulink system	`save_system`
Find a Simulink system or block	`find_system`
Add a new block to a system	`add_block`
Delete a block from a system	`delete_block`
Replace a block in a system	`replace_block`
Add a line to a system	`add_line`
Delete a line from a system	`delete_line`
Get a parameter value	`get_param`
Set parameter values	`set_param`
Get the path of the current block	`gcb`
Get the path of the current system	`gcs`
Get the name of the root-level system	`bdroot`
Open the Simulink block library	`simulink`

How to Specify Parameters for the Commands

The commands described in this chapter require that you specify arguments that describe a system, block, or block parameter. Appendix A provides comprehensive tables of model and block parameters.

How to Specify a Path for a Simulink Object

Many of the commands described in this chapter require that you identify a Simulink system or block. Identify systems and blocks by specifying their paths:

- To identify a system, specify its name, which is the name of the file that contains the system description, without the mdl extension.

  ```
  system
  ```

- To identify a subsystem, specify the system and the hierarchy of subsystems in which the subsystem resides:

  system/subsystem$_1$/.../subsystem

- To identify a block, specify the path of the system that contains the block and specify the block name:

  system/subsystem$_1$/.../subsystem/block

If the block name includes a newline or carriage return, specify the block name as a string vector and use `sprintf('\n')` as the newline character. For example, these lines assign the newline character to `cr`, then get the value for the Signal Generator’s Amplitude parameter:

```
cr = sprintf('\n');
get_param(['untitled/Signal',cr,'Generator'],'Amplitude')
ans =
   1
```

If the block name includes a slash character (/), you repeat the slash when you specify the block name. For example, to get the value of the `Location` parameter for the block named Signal/Noise in the `mymodel` system:

```
get_param('mymodel/Signal//Noise','Location')
```

add_block

Purpose Add a block to a Simulink system.

Syntax

```
add_block('src', 'dest')
add_block('src', 'dest', 'parameter1', value1, ...)
```

Description `add_block('src', 'dest')` copies the block with full path name `'src'` to a new block with full path name `'dest'`. The block parameters of the new block are identical to those of the original. The name `'built-in'` can be used as a source system name for all Simulink built-in blocks (blocks available in Simulink block libraries that are not masked blocks).

`add_block('src', 'dest_obj', 'parameter1', value1, ...)` creates a copy as above, in which the named parameters have the specified values. Any additional arguments must occur in parameter-value pairs.

Examples This command copies the Scope block from the Sinks subsystem of the `simulink` system to a block named `Scope1` in the `timing` subsystem of the `engine` system.

```
add_block('simulink/Sinks/Scope', 'engine/timing/Scope1')
```

This command creates a new subsystem named `controller` in the `F14` system.

```
add_block('built-in/SubSystem', 'F14/controller')
```

This command copies the built-in Gain block to a block named `Volume` in the `mymodel` system and assigns the Gain parameter a value of 4.

```
add_block('built-in/Gain', 'mymodel/Volume', 'Gain', '4')
```

See Also `delete_block`
`set_param`

Purpose

Add a line to a Simulink system.

Syntax

```
add_line('sys', 'oport', 'iport')
add_line('sys', points)
```

Description

The `add_line` command adds a line to the specified system. The line can be defined in two ways:

- By naming the block ports that are to be connected by the line
- By specifying the location of the points that define the line segments

`add_line('sys', 'oport', 'iport')` adds a straight line to a system from the specified block output port `'oport'` to the specified block input port `'iport'`. `'oport'` and `'iport'` are strings consisting of a block name and a port identifier in the form `'block/port'`. Most block ports are identified by numbering the ports from top to bottom or from left to right, such as `'Gain/1'` or `'Sum/2'`. Enable, Trigger, and State ports are identified by name, such as `'subsystem_name/Enable'`, `'subsystem_name/Trigger'`, or `'Integrator/State'`.

`add_line(system, points)` adds a segmented line to a system. Each row of the `points` array specifies the *x* and *y* coordinates of a point on a line segment. The origin is the top left corner of the window. The signal flows from the point defined in the first row to the point defined in the last row. If the start of the new line is close to the output of an existing block or line, a connection is made. Likewise, if the end of the line is close to an existing input, a connection is made.

Examples

This command adds a line to the `mymodel` system connecting the output of the `Sine Wave` block to the first input of the `Mux` block.

```
add_line('mymodel','Sine Wave/1','Mux/1')
```

This command adds a line to the `mymodel` system extending from (`20,55`) to (`40,10`) to (`60,60`).

```
add_line('mymodel',[20 55; 40 10; 60 60])
```

See Also

`delete_line`

bdclose

Purpose

Close any or all Simulink system windows unconditionally.

Syntax

```
bdclose
bdclose('sys')
bdclose('all')
```

Description

`bdclose` with no arguments closes the current system window unconditionally and without confirmation. Any changes made to the system since it was last saved are lost.

`bdclose('sys')` closes the specified system window.

`bdclose('all')` closes all system windows.

Examples

This command closes the `vdp` system:

```
bdclose('vdp')
```

See Also

`close_system`
`new_system`
`open_system`
`save_system`

Purpose Return the name of the top-level Simulink system.

Syntax

```
bdroot
bdroot('obj')
```

Description `bdroot` with no arguments returns the top-level system name.

`bdroot('obj')` where `'obj'` is a system or block path name, returns the name of the top-level system containing the specified object name.

Examples This command returns the name of the top-level system that contains the current block:

```
bdroot(gcb)
```

See Also `find_system`
`gcb`

close_system

Purpose Close a Simulink system window or a block dialog box.

Syntax

```
close_system
close_system('sys')
close_system('sys', saveflag)
close_system('sys', 'newname')
close_system('blk')
```

Description `close_system` with no arguments closes the current system or subsystem window. If the current system is the top-level system and it has been modified, then `close_system` asks if the changed system should be saved to a file before removing the system from memory. The current system is defined in the description of the `gcs` command, on page 11-16.

`close_system('sys')` closes the specified system or subsystem window.

`close_system('sys', saveflag)` closes the specified top-level system window and removes it from memory.

- If `saveflag` is 0, the system is not saved.
- If `saveflag` is 1, the system is saved with its current name.

`close_system('sys', 'newname')` saves the specified top-level system to a file with the specified new name, then closes the system.

`close_system('blk')` where `'blk'` is a full block path name, closes the dialog box associated with the specified block or calls the block's `CloseFcn` callback parameter if one is defined. Any additional arguments are ignored.

Examples This command closes the current system:

```
close_system
```

This command closes the `vdp` system:

```
close_system('vdp')
```

This command saves the `engine` system with its current name, then closes it:

```
close_system('engine', 1)
```

This command closes the `mymdl12` system with the name `testsys`, then closes it:

```
close_system('mymdl12', 'testsys')
```

This command closes the dialog box of the Unit Delay block in the Combustion subsystem of the `engine` system:

```
close_system('engine/Combustion/Unit Delay')
```

See Also

`bdclose`
`gcs`
`new_system`
`open_system`
`save_system`

delete_block

Purpose Delete a block from a Simulink system.

Syntax

```
delete_block('blk')
```

Description `delete_block('blk')` where `'blk'` is a full block path name, deletes the specified block from a system.

Example This command removes block Out1 from the `vdp` system:

```
delete_block('vdp/Out1')
```

See Also `add_block`

Purpose Delete a line from a Simulink system.

Syntax

```
delete_line('sys', 'oport', 'iport')
```

Description `delete_line('sys', 'oport', 'iport')` deletes the line extending from the specified block output port `'oport'` to the specified block input port `'iport'`. `'oport'` and `'iport'` are strings consisting of a block name and a port identifier in the form `'block/port'`. Most block ports are identified by numbering the ports from top to bottom or from left to right, such as `'Gain/1'` or `'Sum/2'`. Enable, Trigger, and State ports are identified by name, such as `'subsystem_name/Enable'`, `'subsystem_name/Trigger'`, or `'Integrator/State'`.

`delete_line('sys', [x y])` deletes one of the lines in the system that contains the specified point (x,y), if any such line exists.

Example This command removes the line from the `mymodel` system connecting the Sum block to the second input of the Mux block.

```
delete_line('mymodel','Sum/1','Mux/2')
```

See Also `add_line`

find_system

Purpose

Find Simulink objects with specified parameter values.

Syntax

```
find_system('parameter1', value1, 'parameter2', value2,...)
find_system('SearchDepth', depth, 'parameter1', value1, ...)
find_system('obj', ...)
find_system(names, ...)
find_system('LookUnderMasks', 'on', ...)
find_system('FollowLinks', 'on', ...)
```

Description

`find_system('parameter1', value1, 'parameter2', value2,...)` searches all open systems and returns a cell array containing the full path names in hierarchical order of all systems, subsystems, and blocks whose specified parameters have the specified values. Searches do not extend into masked systems or library blocks (but see the last two forms of the command). Case is ignored for parameter names. Value strings are case sensitive. Any parameters that correspond to dialog box entries have string values. See Appendix A for a list of model and block parameters.

`find_system('SearchDepth', depth, parameter, value, ...)` restricts the search to a specified depth from the top-level system The scalar `depth` argument is:

- 1 for blocks and systems contained in the top-level system.
- 2 for blocks and systems contained in the top-level system and in the children of that system, and so on.

`find_system('obj', ...)`, where `'obj'` is a system or block path name, searches only in the specified object.

`find_system(names)`, where `names` is a cell array of system or block path names, searches only the objects listed in `names`. This allows the search to be restricted to objects found by previous `find_system` calls.

`find_system('LookUnderMasks', 'on', ...)` enables you to search within a masked system (unless the mask consists only of an icon). If the default value, `'off'`, is in effect, the search does not extend into masked systems.

`find_system('FollowLinks', 'on', ...)` enables you to follow a link into a library. If the default value, `'off'`, is in effect, the search does not extend into linked blocks.

Examples

This command returns a cell array containing the names of all open systems and blocks:

```
find_system
```

This command returns the names of all open block diagrams:

```
open_bd = find_system('Type', 'block_diagram')
```

This command returns the names of all Goto blocks that are children of the `Unlocked` subsystem in the `clutch` system:

```
find_system('clutch/Unlocked','SearchDepth',1,'BlockType','Goto')
```

These commands return the names of all Gain blocks in the `vdp` system having a Gain parameter value of 1.

```
gb = find_system('vdp', 'BlockType', 'Gain')
find_system(gb, 'Gain', '1')
```

The above commands are equivalent to this command:

```
find_system('vdp', 'BlockType', 'Gain', 'Gain', '1')
```

See Also

`get_param`
`set_param`

gcb

Purpose

Get the full block path name of the current Simulink block.

Syntax

```
gcb
gcb('sys')
```

Description

`gcb` returns the full block path name of the current block in the current system.

`gcb('sys')` returns the full block path name of the current block in the specified system.

The current block is one of these:

- During editing, the current block is the block most recently clicked on.
- During simulation of a system that contains S-Function blocks, the current block is the S-Function block currently executing its corresponding MATLAB function.
- During callbacks, the current block is the block whose callback routine is being executed.
- During evaluation of the `MaskInitialization` string, the current block is the block whose mask is being evaluated.

Examples

This command returns the path of the most recently selected block.

```
>> gcb
ans =
   clutch/Locked/Inertia
```

This command gets the value of the Gain parameter of the current block:

```
>> get_param(gcb,'Gain')
ans =
   1/(Iv+Ie)
```

See Also

`gcbh`, `gcs`

Purpose Get the handle of the current Simulink block.

Syntax

```
gcbh
```

Description `gcbh` returns the handle of the current block in the current system.

You can use this command to identify or address blocks that have no parent system. The command should be most useful to blockset authors.

Examples This command returns the handle of the most recently selected block.

```
>> gcbh

ans =

   281.0001
```

See Also `gcb`

gcs

Purpose Get the full path name of the current Simulink system.

Syntax

```
gcs
```

Description `gcs` returns the full path name of the current system.

The current system is:

- During editing, the current system is the system or subsystem most recently clicked in.
- During simulation of a system that contains S-Function blocks, the current system is the system or subsystem containing the S-Function block that is currently being evaluated.
- During callbacks, the current system is the system containing any block whose callback routine is being executed.
- During evaluation of the `MaskInitialization` string, the current system is the system containing the block whose mask is being evaluated.

Examples This example returns the path of the system that contains the most recently selected block.

```
>> gcs
ans =
   clutch/Locked
```

See Also `gcb`

Purpose

Get Simulink system and block parameter values.

Syntax

```
get_param('obj', 'parameter')
get_param( { objects }, 'parameter')
```

Description

`get_param('obj', 'parameter')`, where `'obj'` is a system or block path name, returns the value of the specified parameter. Case is ignored for parameter names.

`get_param( { objects }, 'parameter')` accepts a cell array of full path specifiers, enabling you to get the values of a parameter common to all objects specified in the cell array.

Appendix A contains lists of model and block parameters.

Examples

This command returns the value of the Gain parameter for the Inertia block in the `Requisite Friction` subsystem of the `clutch` system.

```
get_param('clutch/Requisite Friction/Inertia','Gain')
ans =
   1/(Iv+Ie)
```

These commands display the block types of all blocks in the `mx+b` system (the current system), described in "A Sample Masked Subsystem" on page 6–3:

```
>> blks = find_system(gcs, 'Type', 'block');
>> listblks = get_param(blks, 'BlockType')

listblks =

    'SubSystem'
    'Inport'
    'Constant'
    'Gain'
    'Sum'
    'Outport'
```

See Also

`find_system`
`set_param`

new_system

Purpose Create a new empty Simulink system.

Syntax

```
new_system('sys')
```

Description `new_system('sys')` creates a new empty system with the specified name. If `'sys'` specifies a path, the new system will be a subsystem of the system specified in the path. `new_system` does not open the system window.

For a list of the default parameter values for the new system, see Appendix A.

Example This command creates a new system named `'mysys'`:

```
new_system('mysys')
```

This command creates a new subsystem named `'mysys'` in the `vdp` system:

```
new_system('vdp/mysys')
```

See Also

`close_system`
`open_system`
`save_system`

Purpose Open a Simulink system window or a block dialog box.

Syntax

```
open_system('sys')
open_system('blk')
open_system('blk', 'force')
```

Description `open_system('sys')` opens the specified system or subsystem window.

`open_system('blk')`, where `'blk'` is a full block path name, opens the dialog box associated with the specified block. If the block's `OpenFcn` callback parameter is defined, the routine is evaluated.

`open_system('blk', 'force')`, where `'blk'` is a full path name or a masked system, looks under the mask of the specified system. This command is equivalent to using the **Look Under Mask** menu item.

Example This command opens the `controller` system in its default screen location:

```
open_system('controller')
```

This command opens the block dialog box for the Gain block in the `controller` system:

```
open_system('controller/Gain')
```

See Also

`close_system`
`new_system`
`save_system`

replace_block

Purpose

Replace blocks in a Simulink model.

Syntax

```
replace_block('sys', 'blk1', 'blk2', 'noprompt')
replace_block('sys', 'Parameter', 'value', 'blk', ...)
```

Description

`replace_block('sys', 'blk1', 'blk2')` replaces all blocks in `'sys'` having the block or mask type `'blk1'` with `'blk2'`. If `'blk2'` is a Simulink built-in block, only the block name is necessary. If `'blk'` is in another system, its full block path name is required. If `'noprompt'` is omitted, Simulink displays a dialog box that asks you to select matching blocks before making the replacement. Specifying the `'noprompt'` argument suppresses the dialog box from being displayed. If a return variable is specified, the paths of the replaced blocks are stored in that variable.

`replace_block('sys', 'Parameter', 'value', ..., 'blk')` replaces all blocks in `'sys'` having the specified values for the specified parameters with `'blk'`. You can specify any number of parameter/value pairs.

NOTE Because it may be difficult to undo the changes this command makes, it is a good idea to save your system first.

Example

This command replaces all Gain blocks in the `f14` system with Integrator blocks and stores the paths of the replaced blocks in `RepNames`. Simulink lists the matching blocks in a dialog box before making the replacement.

```
RepNames = replace_block('f14','Gain','Integrator')
```

This command replaces all blocks in the Unlocked subsystem in the `clutch` system having a Gain of `'bv'` with the Integrator block. Simulink displays a dialog box listing the matching blocks before making the replacement.

```
replace_block('clutch/Unlocked','Gain','bv','Integrator')
```

This command replaces the Gain blocks in the `f14` system with Integrator blocks but does not display the dialog box:

```
replace_block('f14','Gain','Integrator','noprompt')
```

See Also

`find_system`, `set_param`

Purpose

Save a Simulink system.

Syntax

```
save_system
save_system('sys')
save_system('sys', 'newname')
```

Description

`save_system` saves the current top-level system to a file with its current name.

`save_system('sys')` saves the specified top-level system to a file with its current name. The system must be open.

`save_system('sys', 'newname')` saves the specified top-level system to a file with the specified new name. The system must be open.

Example

This command saves the current system:

```
save_system
```

This command saves the `vdp` system:

```
save_system('vdp')
```

This command saves the `vdp` system to a file with the name `'myvdp'`:

```
save_system('vdp', 'myvdp')
```

See Also

`close_system`
`new_system`
`open_system`

set_param

Purpose Set Simulink system and block parameters.

Syntax

```
set_param('obj', 'parameter1', value1, 'parameter2', value2, ...)
```

Description

`set_param('obj', 'parameter1', value1, 'parameter2', value2, ...)`, where `'obj'` is a system or block path, sets the specified parameters to the specified values. Case is ignored for parameter names. Value strings are case sensitive. Any parameters that correspond to dialog box entries have string values. Model and block parameters are listed in Appendix A.

You can change block parameter values in the workspace during a simulation and update the block diagram with these changes. To do this, make the changes in the command window, then make the model window the active window, then choose **Update Diagram** from the **Edit** menu.

NOTE Most block parameter values must be specified as strings. Two exceptions are the `Position` and `UserData` parameters, common to all blocks.

Examples

This command sets the `Solver` and `StopTime` parameters of the `vdp` system:

```
set_param('vdp', 'Solver', 'ode15s', 'StopTime', '3000')
```

This command sets the Gain of block Mu in the `vdp` system to 1000 (stiff):

```
set_param('vdp/Mu', 'Gain', '1000')
```

This command sets the position of the Fcn block in the `vdp` system:

```
set_param('vdp/Fcn', 'Position', [50 100 110 120])
```

This command sets the `Zeros` and `Poles` parameters for the Zero-Pole block in the `mymodel` system:

```
set_param('mymodel/Zero-Pole','Zeros','[2 4]','Poles','[1 2 3]')
```

This command sets the `Gain` parameter for a block in a masked subsystem. The variable `k` is associated with the `Gain` parameter:

```
set_param('mymodel/Subsystem', 'k', '10')
```

This command sets the OpenFcn callback parameter of the block named Compute in system mymodel. The function 'my_open_fcn' executes when the user double-clicks on the Compute block. For more information, see “Using Callback Routines” on page 3–30.

```
set_param('mymodel/Compute', 'OpenFcn', 'my_open_fcn')
```

See Also

get_param
find_system

Purpose

Open the Simulink block library.

Syntax

```
simulink
```

Description

The `simulink` command opens the Simulink block library window and creates and displays a new (empty) model window, except in these cases:

- If a model window is open, Simulink does not create a new model window.
- If the Simulink block library window is already open, issuing this command makes the Simulink window the active window.

A

Model and Block Parameters

Introduction

This appendix lists model, block, and mask parameters. The tables that list the parameters provide enough information to enable you to modify models from the command line, using the set_param command, described in Chapter 11.

Model Parameters

This table lists and describes parameters that describe a model. The parameters appear in the order they are defined in the model file, described in Appendix B. The table also includes model callback parameters, described in “Using Callback Routines” on page 3–30. The **Description** column indicates where you can set the value on the **Simulation Parameters** dialog box. Model parameters that are simulation parameters are described in more detail in Chapter 4. Examples showing how to change parameters follow the table.

Parameter values must be specified as quoted strings. The string contents depend on the parameter and can be numeric (scalar, vector, or matrix), a variable name, a file name, or a particular value. The **Values** column shows the type of value required, the possible values (separated with a vertical line), and the default value, enclosed in braces.

Table A-1: Model Parameters

Parameter	Description	Values
`Name`	Model name	text
`Version`	Simulink version used to modify the model (read-only)	(release)
`SimParamPage`	**Simulation Parameters** dialog box page to display (page last displayed)	`{Solver} | WorkspaceI/O | Diagnostics`
`SampleTimeColors`	**Sample Time Colors** menu option	`on | {off}`
`InvariantConstants`	Invariant constant setting	`on | {off}`
`WideVectorLines`	**Wide Vector Lines** menu option	`on | {off}`
`ShowLineWidths`	**Show Line Widths** menu option	`on | {off}`
`PaperOrientation`	Printing paper orientation	`portrait | {landscape}`
`PaperType`	Printing paper type	`{usletter} | uslegal | a3 | a4 | a5 | b4 | tabloid`
`PaperUnits`	Printing paper size units	`normalized | {inches} | centimeters | points`
`StartTime`	Simulation start time	scalar `{0.0}`

Table A-1: Model Parameters (Continued)

Parameter	Description	Values
StopTime	Simulation stop time	scalar {10.0}
Solver	Solver	{ode45} | ode23 | ode113 | ode15s | ode23s | ode5 | ode4 | ode3 | ode2 | ode1 | discrete
RelTol	Relative error tolerance	scalar {1e–3}
AbsTol	Absolute error tolerance	scalar {1e–6}
Refine	Refine factor	scalar {1}
MaxStep	Maximum step size	scalar {auto}
InitialStep	Initial step size	scalar {auto}
FixedStep	Fixed step size	scalar {auto}
MaxOrder	Maximum order for ode15s	1 | 2 | 3 | 4 | {5}
OutputOption	Output option	AdditionalOutputTimes | {RefineOutputTimes} | SpecifiedOutputTimes
OutputTimes	Values for chosen OutputOption	vector {[]}
LoadExternalInput	Load input from workspace	on | {off}
ExternalInput	Time and input variable names	scalar or vector [t, u]
SaveTime	Save simulation time	{on} | off
TimeSaveName	Simulation time name	variable {tout}
SaveState	Save states	on | {off}
StateSaveName	State output name	variable {xout}
SaveOutput	Save simulation output	{on} | off
OutputSaveName	Simulation output name	variable {yout}
LoadInitialState	Load initial state	on | {off}

Table A-1: Model Parameters (Continued)

Parameter	Description	Values
`InitialState`	Initial state name or values	variable or vector `{xInitial}`
`SaveFinalState`	Save final state	`on	{off}`
`FinalStateName`	Final state name	variable `{xFinal}`
`LimitMaxRows`	Limit output	`on	{off}`
`MaxRows`	Maximum number of output rows to save	scalar `{1000}`
`Decimation`	Decimation factor	scalar `{1}`
`AlgebraicLoopMsg`	Algebraic loop diagnostic	`none	{warning}	error`
`MinStepSizeMsg`	Minimum step size diagnostic	`{warning}	error`
`UnconnectedInputMsg`	Unconnected input ports diagnostic	`none	{warning}	error`
`UnconnectedOutputMsg`	Unconnected output ports diagnostic	`none	{warning}	error`
`UnconnectedLineMsg`	Unconnected lines diagnostic	`none	{warning}	error`
`ConsistencyChecking`	Consistency checking	`on	{off}`
`ZeroCross`	Intrinsic zero crossing detection (see "Zero Crossings" on page 10–3)	`{on}	off`
`CloseFcn`	Close callback	command or variable
`PreLoadFcn`	Pre-load callback	command or variable
`PostLoadFcn`	Post-load callback	command or variable
`SaveFcn`	Save callback	command or variable
`StartFcn`	Start simulation callback	command or variable
`StopFcn`	Stop simulation callback	command or variable

These examples show how to set model parameters for the `mymodel` system.

This command sets the simulation start and stop times:

```
set_param('mymodel','StartTime','5','StopTime','100')
```

This command sets the solver to `ode15s` and changes the maximum order:

```
set_param('mymodel','Solver','ode15s','MaxOrder','3')
```

This command associates a `SaveFcn` callback:

```
set_param('mymodel','SaveFcn','my_save_cb')
```

Common Block Parameters

This table lists the parameters common to all Simulink blocks, including block callback parameters, which are described in “Using Callback Routines” on page 3–30. Examples of commands that change these parameters follow the table.

Table A-2: Common Block Parameters

Parameter	Description	Values
`Name`	Block’s name	string
`Type`	Simulink object type (read-only)	`'block'`
`Parent`	Name of the system that owns the block	string
`BlockType`	Block type	text
`BlockDescription`	Block description	text
`InputPorts`	Array of input port locations	`[h1,v1; h2,v2; ...]`
`OutputPorts`	Array of output port locations	`[h1,v1; h2,v2; ...]`
`Orientation`	Where block faces	`{right} | left | down | up`
`ForegroundColor`	Block name, icon, outline, output signals, and signal label	`{black} | white | red | green | blue | cyan | magenta | yellow | gray | lightBlue | orange | darkGreen`
`BackgroundColor`	Block icon background	`black | {white} | red | green | blue | cyan | magenta | yellow | gray | lightBlue | orange | darkGreen`
`DropShadow`	Display drop shadow	`{off} | on`
`NamePlacement`	Position of block name	`{normal} | alternate`
`FontName`	Font	`{Helvetica}`
`FontSize`	Font size	`{10}`

Table A-2: Common Block Parameters (Continued)

Parameter	Description	Values
`FontWeight`	Font weight	(system-dependent) `light | {normal} | demi | bold`
`FontAngle`	Font angle	(system-dependent) `{normal} | italic | oblique`
`Position`	Position of block in model window	vector `[left top right bottom]` NOT enclosed in quotes
`ShowName`	Display block name	`{on} | off`
`Tag`	User-defined string	`''`
`UserData`	Any MATLAB data type (not saved in the `mdl` file)	`[]`
`Seleted`	Block selected state	`on | {off}`
`CloseFcn`	Close callback	MATLAB expression
`CopyFcn`	Copy callback	MATLAB expression
`DeleteFcn`	Delete callback	MATLAB expression
`InitFcn`	Initialization callback	MATLAB expression
`LoadFcn`	Load callback	MATLAB expression
`ModelCloseFcn`	Model close callback	MATLAB expression
`NameChangeFcn`	Block name change callback	MATLAB expression
`OpenFcn`	Open callback	MATLAB expression
`ParentCloseFcn`	Parent subsystem close call-back	MATLAB expression
`PreSaveFcn`	Pre-save callback	MATLAB expression
`PostSaveFcn`	Post-save callback	MATLAB expression
`StartFcn`	Start simulation callback	MATLAB expression

Table A-2: Common Block Parameters (Continued)

Parameter	Description	Values
`StopFcn`	Termination of simulation callback	MATLAB expression
`UndoDeleteFcn`	Undo block delete callback	MATLAB expression

These examples illustrate how to change these parameters.

This command changes the orientation of the Gain block in the `mymodel` system so it faces the opposite direction (right to left):

```
set_param('mymodel/Gain','Orientation','left')
```

This command associates an `OpenFcn` callback with the Gain block in the `mymodel` system:

```
set_param('mymodel/Gain','OpenFcn','my_open_cb')
```

This command sets the `Position` parameter of the Gain block in the `mymodel` system. The block is 75 pixels wide by 25 pixels high. The position vector is *not* enclosed in quotes.

```
set_param('mymodel/Gain','Position',[50 250 125 275])
```

Block-Specific Parameters

These tables list block-specific parameters for all Simulink blocks. When setting block parameters with the `set_param` command, you identify the block by specifying its `BlockType` parameter. The `BlockType` appears in parentheses after the block name.

The table includes detailed information only for built-in blocks, not for masked blocks, although the table includes the `MaskType` parameter value for masked blocks. For more information, see "Mask Parameters" on page A–23.

The **Dialog Box Prompt** column indicates the text of the prompt for the parameter on the block's dialog box. The **Values** column shows the type of value required (scalar, vector, variable), the possible values (separated with a vertical line), and the default value (enclosed in braces).

Table A-3: Connections Library Block Parameters

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Data Store Memory (`DataStoreMemory`)		
`DataStoreName`	Data store name	`tag {A}`
`InitialValue`	Initial value	vector `{0}`
Data Store Read (`DataStoreRead`)		
`DataStoreName`	Data store name	`tag {A}`
`SampleTime`	Sample time	scalar (sample period) `{-1}` or vector [period offset]
Data Store Write (`DataStoreWrite`)		
`DataStoreName`	Data store name	`tag {A}`
`SampleTime`	Sample time	scalar (sample period) `{-1}` or vector [period offset]
Demux (`Demux`)		
`Outputs`	Number of outputs	scalar or vector `{3}`

Table A-3: Connections Library Block Parameters (Continued)

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Enable (EnablePort)		
StatesWhenEnabling	States when enabling	{held} | reset
ShowOutputPort	Show output port	{off} | on
From (From)		
GotoTag	Goto tag	tag {A}
Goto (Goto)		
GotoTag	Tag	tag {A}
TagVisibility	Tag visibility	{local} | scoped | global
Goto Tag Visibility (GotoTagVisibility)		
GotoTag	Goto tag	tag {A}
Ground (Ground) (no block-specific parameters)		
IC (InitialCondition)		
Value	Initial value	scalar or vector {1}
In (Inport)		
Port	Port number	scalar {1}
PortWidth	Port width	scalar {-1}
SampleTime	Sample time	scalar (sample period) {-1} or vector [period offset]
Mux (Mux)		
Inputs	Number of inputs	scalar or vector {3}
Out (Outport)		
Port	Port number	scalar {1}
OutputWhenDisabled	Output when disabled	{held} | reset
InitialOutput	Initial output	scalar or vector {0}

Table A-3: Connections Library Block Parameters (Continued)

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Selector (`Selector`)		
`Elements`	Elements	vector `{[1 3]}`
`InputPortWidth`	Input port width	scalar `{3}`
Subsystem (`SubSystem`)		
`ShowPortLabels`	Show/Hide Port Labels **Format** menu item	`{on} | off`
Terminator (`Terminator`) (no block-specific parameters)		
Trigger (`TriggerPort`)		
`TriggerType`	Trigger type	`{rising} | falling | either | function-call`
`ShowOutputPort`	Show output port	`{off} | on`
Width (`Width`) (no block-specific parameters)		

Table A-4: Discrete Library Block Parameters

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Discrete Filter (`DiscreteFilter`)		
`Numerator`	Numerator	vector `{[1]}`
`Denominator`	Denominator	vector `{[1 2]}`
`SampleTime`	Sample time	scalar (sample period) `{1}` or vector [period offset]
Discrete-Time Integrator (`DiscreteIntegrator`)		
`IntegratorMethod`	Integrator method	`{ForwardEuler} | BackwardEuler | Trapezoidal`
`ExternalReset`	External reset	`{none} | rising | falling | either`

Table A-4: Discrete Library Block Parameters (Continued)

Block (BlockType)/Parameter	Dialog Box Prompt	Values
`InitialConditionSource`	Initial condition source	`{internal} | external`
`InitialCondition`	Initial condition	scalar or vector `{0}`
`LimitOutput`	Limit output	`{off} | on`
`UpperSaturationLimit`	Upper saturation limit	scalar or vector `{inf}`
`LowerSaturationLimit`	Lower saturation limit	scalar or vector `{-inf}`
`ShowSaturationPort`	Show saturation port	`{off} | on`
`ShowStatePort`	Show state port	`{off} | on`
`SampleTime`	Sample time	scalar (sample period) `{1}` or vector [period offset]
Discrete State-Space (`DiscreteStateSpace`)		
`A`	A	matrix `{1}`
`B`	B	matrix `{1}`
`C`	C	matrix `{1}`
`D`	D	matrix `{1}`
`X0`	Initial conditions	vector `{0}`
`SampleTime`	Sample time	scalar (sample period) `{1}` or vector [period offset]
Discrete Transfer Fcn (`DiscreteTransferFcn`)		
`Numerator`	Numerator	vector `{[1]}`
`Denominator`	Denominator	vector `{[1 0.5]}`
`SampleTime`	Sample time	scalar (sample period) `{1}` or vector [period offset]
Discrete Zero-Pole (`DiscreteZeroPole`)		
`Zeros`	Zeros	vector `{[1]}`

Table A-4: Discrete Library Block Parameters (Continued)

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Poles	Poles	vector [0 0.5]
Gain	Gain	scalar {1}
SampleTime	Sample time	scalar (sample period) {1} or vector [period offset]
First-Order Hold (First Order Hold) (masked)		
Unit Delay (UnitDelay)		
X0	Initial condition	scalar or vector {0}
SampleTime	Sample time	scalar (sample period) {1} or vector [period offset]
Zero-Order Hold (ZeroOrderHold)		
SampleTime	Sample time	scalar (sample period) {1} or vector [period offset]

Table A-5: Linear Library Block Parameters

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Derivative (Derivative) (no block-specific parameters)		
Dot Product (Dot Product) (masked)		
Gain (Gain)		
Gain	Gain	scalar or vector {1}
Integrator (Integrator)		
ExternalReset	External reset	{none} | rising | falling | either
InitialConditionSource	Initial condition source	{internal} | external
InitialCondition	Initial condition	scalar or vector {0}
LimitOutput	Limit output	{off} | on

Table A-5: Linear Library Block Parameters (Continued)

Block (BlockType)/Parameter	Dialog Box Prompt	Values
`UpperSaturationLimit`	Upper saturation limit	scalar or vector `{inf}`
`LowerSaturationLimit`	Lower saturation limit	scalar or vector `{-inf}`
`ShowSaturationPort`	Show saturation port	`{off} | on`
`ShowStatePort`	Show state port	`{off} | on`
`AbsoluteTolerance`	Absolute tolerance	scalar `{auto}`
Matrix Gain (`Matrix Gain`) (masked)		
Slider Gain (`SliderGain`) (masked)		
State-Space (`StateSpace`)		
`A`	A	matrix `{1}`
`B`	B	matrix `{1}`
`C`	C	matrix `{1}`
`D`	D	matrix `{1}`
`X0`	Initial conditions	vector `{0}`
Sum (`Sum`)		
`Inputs`	List of signs	scalar or list of signs `{++}`
Transfer Fcn (`TransferFcn`)		
`Numerator`	Numerator	vector or matrix `{[1]}`
`Denominator`	Denominator	vector `{[1 1]}`
Zero-Pole (`ZeroPole`)		
`Zeros`	Zeros	vector `{[1]}`
`Poles`	Poles	vector `{[0 -1]}`
`Gain`	Gain	vector `{[1]}`

Table A-6: Nonlinear Library Block Parameters

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Abs (`Abs`) (no block-specific parameters)		
Algebraic Constraint (`Algebraic Constraint`) (masked)		
Backlash (`Backlash`)		
`BacklashWidth`	Deadband width	scalar or vector `{1}`
`InitialOutput`	Initial output	scalar or vector `{0}`
Combinatorial Logic (`CombinatorialLogic`)		
`TruthTable`	Truth table	matrix `{[0 0;0 1;0 1;1 0; 0 1;1 0;1 0;1 1]}`
Coulomb & Viscous Friction (`Coulombic and Viscous Friction`) (masked)		
Dead Zone (`DeadZone`)		
`LowerValue`	Start of dead zone	scalar or vector `{-0.5}`
`UpperValue`	End of dead zone	scalar or vector `{0.5}`
Elementary Math (`ElementaryMath`)		
`Operator`	Operator	`{sin} | cos | tan | asin | acos | atan | atan2 | sinh | cosh | tanh | exp | log | log10 | floor | ceil | sqrt | reciprocal | pow | hypot`
Fcn (`Fcn`)		
`Expn`	Expression	expression `{sin(u(1)*exp(2.3*(-u(2))))}`
Hit Crossing (`HitCross`)		
`HitCrossingOffset`	Hit crossing offset	scalar or vector `{0}`
`HitCrossingDirection`	Hit crossing direction	`rising | falling | {either}`
`ShowOutputPort`	Show output port	`{on} | off`
Logical Operator (`Logic`)		

Table A-6: Nonlinear Library Block Parameters (Continued)

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Operator	Operator	{AND} | OR | NAND | NOR | XOR | NOT
Inputs	Number of input ports	scalar {2}
Look-up Table (Lookup)		
InputValues	Vector of input values	vector {[-5:5]}
OutputValues	Vector of output values	vector {tanh([-5:5])}
Look-Up Table (2-D) (Lookup Table (2-D))		
RowIndex	Row	vector
ColumnIndex	Column	vector
OutputValues	Table	2-D matrix
Math Function (Math)		
Operator	Function	{exp} | log | log10 | square | sqrt | pow | reciprocal | hypot | rem | mod
MATLAB Fcn (MATLABFcn)		
MATLABFcn	MATLAB function	MATLAB function {sin}
OutputWidth	Output width	scalar or vector {-1}
Memory (Memory)		
X0	Initial condition	scalar or vector {0}
InheritSampleTime	Inherit sample time	{off} | on
MinMax (MinMax)		
Function	Function	{min} | max
Inputs	Number of input ports	scalar {1}
Multiport Switch (MultiPortSwitch)		
Inputs	Number of inputs	scalar or vector {3}

Table A-6: Nonlinear Library Block Parameters (Continued)

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Product (`Product`)		
`Inputs`	Number of inputs	scalar `{2}`
Quantizer (`Quantizer`)		
`QuantizationInterval`	Quantization interval	scalar or vector `{0.5}`
Rate Limiter (`RateLimiter`)		
`RisingSlewLimit`	Rising slew rate	scalar or vector `{1.}`
`FallingSlewLimit`	Falling slew rate	scalar or vector `{-1.}`
Relational Operator (`RelationalOperator`)		
`Operator`	Operator	`== | != | < | {<=} | >= | >`
Relay (`Relay`)		
`OnSwitchValue`	Switch on point	scalar or vector `{eps}`
`OffSwitchValue`	Switch off point	scalar or vector `{eps}`
`OnOutputValue`	Output when on	scalar or vector `{1}`
`OffOutputValue`	Output when off	scalar or vector `{0}`
Rounding Function (`Rounding`)		
`Operator`	Function	`{floor} | ceil | round | fix`
Saturation (`Saturate`)		
`UpperLimit`	Upper limit	scalar or vector `{0.5}`
`LowerLimit`	Lower limit	scalar or vector `{-0.5}`
S-Function (`S-Function`)		
`FunctionName`	S-function name	name `{system}`
`Parameters`	S-function parameters	additional parameters if needed
Sign (`Signum`) (no block-specific parameters)		

Table A-6: Nonlinear Library Block Parameters (Continued)

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Switch (`Switch`)		
`Threshold`	Threshold	scalar or vector `{0}`
Transport Delay (`TransportDelay`)		
`DelayTime`	Time delay	scalar or vector `{1}`
`InitialInput`	Initial input	scalar or vector `{0}`
`BufferSize`	Initial buffer size	scalar `{1024}`
Trigonometric Function (`Trigonometry`)		
`Operator`	Function	`{sin} | cos | tan | asin | acos | atan | atan2 | sinh | cosh | tanh`
Variable Transport Delay (`VariableTransportDelay`)		
`MaximumDelay`	Maximum delay	scalar or vector `{10}`
`InitialInput`	Initial input	scalar or vector `{0}`
`MaximumPoints`	Buffer size	scalar `{1024}`

Table A-7: Sinks Library Block Parameters

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Display (`Display`)		
`Format`	Format	`{short} | long | short_e | long_e | bank`
`Decimation`	Decimation	scalar `{1}`
`Floating`	Floating display	`{off} on`
`SampleTime`	Sample time	scalar (sample period) `{-1}` or vector [period offset]
Scope (`Scope`)		

Table A-7: Sinks Library Block Parameters (Continued)

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Open	(If Scope open when the model is opened. Cannot set from dialog box)	{off} | on
Grid	(for future use)	{on} | off
TickLabels	Hide tick labels	{on} | off
ZoomMode	(Zoom button initially pressed)	{on} | xonly | yonly
TimeRange	Time range	scalar {auto}
YMin	Y min	scalar {-5}
YMax	Y max	scalar {5}
SaveToWorkspace	Save data to workspace	{off} | on
SaveName	Variable name	variable {ScopeData}
LimitMaxRows	Limit rows to last	{on} | off
MaxRows	(no label)	scalar {5000}
Decimation	(Value if Decimation selected)	scalar {1}
SampleInput	(Toggles with Decimation)	{off} | on
SampleTime	(SampleInput value)	scalar (sample period) {0} or vector [period offset]
Stop Simulation (StopSimulation) (no block-specific parameters)		
To File (ToFile)		
Filename	Filename	file name {untitled.mat}
MatrixName	Variable name	variable {ans}
Decimation	Decimation	scalar {1}

Table A-7: Sinks Library Block Parameters (Continued)

Block (BlockType)/Parameter	Dialog Box Prompt	Values
`SampleTime`	Sample time	scalar (sample period) `{-1}` or vector [period offset]
To Workspace (`ToWorkspace`)		
`VariableName`	Variable name	variable `{simout}`
`Buffer`	Maximum number of rows	scalar `{inf}`
`Decimation`	Decimation	scalar `{1}`
`SampleTime`	Sample time	scalar (sample period) `{-1}` or vector [period offset]
XY Graph (`XY scope.`) (masked)		

Table A-8: Sources Library Block Parameters

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Band-Limited White Noise (`Continuous White Noise`) (masked)		
Chirp Signal (`chirp`) (masked)		
Clock (`Clock`) (no block-specific parameters)		
Constant (`Constant`)		
`Value`	Constant value	scalar or vector `{1}`
Digital Clock (`DigitalClock`)		
`SampleTime`	Sample time	scalar (sample period) `{1}` or vector [period offset]
From File (`FromFile`)		
`FileName`	File name	file name `{untitled.mat}`
From Workspace (`FromWorkspace`)		

Table A-8: Sources Library Block Parameters (Continued)

Block (BlockType)/Parameter	Dialog Box Prompt	Values
VariableName	Matrix table	matrix {[T,U]}
Pulse Generator (Pulse Generator) (masked)		
Ramp (Ramp) (masked)		
Random Number (RandomNumber)		
Seed	Initial seed	scalar or vector {0}
Repeating Sequence (Repeating table) (masked)		
Signal Generator (SignalGenerator)		
WaveForm	Wave form	{sine} | square | sawtooth | random
Amplitude	Amplitude	scalar or vector {1}
Frequency	Frequency	scalar or vector {1}
Units	Units	{Hertz} | rad/sec
Sine Wave (Sin)		
Amplitude	Amplitude	scalar or vector {1}
Frequency	Frequency	scalar or vector {1}
Phase	Phase	scalar or vector {0}
SampleTime	Sample time	scalar (sample period) {–1} or vector [period offset]
Step (Step)		
Time	Step time	scalar or vector {1}
Before	Initial value	scalar or vector {0}
After	Final value	scalar or vector {1}

Mask Parameters

This section lists parameters that describe masked blocks. This table lists masking parameters, which correspond to **Mask Editor** dialog box parameters.

Table A-9: Mask Parameters

Parameter	Dialog Box Parameter	Values
MaskType	Mask type	string
MaskDescription	Block description	string
MaskHelp	Block help	string
MaskPrompts	Prompt (see below)	cell array of strings
MaskPromptString	Prompt (see below)	delimited string
MaskStyles	Control type (see below)	cell array {Edit} | Checkbox | Popup
MaskStyleString	Control type (see below)	{Edit} | Checkbox | Popup
MaskVariables	Variable (see below)	string
MaskInitialization	Initialization commands	MATLAB command
MaskDisplay	Drawing commands	display commands
MaskIconFrame	Icon frame (Visible is on, Invisible is off)	{on} | off
MaskIconOpaque	Icon transparency (Opaque is on, Transparent is off)	{on} | off
MaskIconRotate	Icon rotation (Rotates is on, Fixed is off)	on | {off}
MaskIconUnits	Drawing coordinates	Pixel | {Autoscale} | Normalized
MaskValues	Block parameter values (see below)	cell array of strings
MaskValueString	Block parameter values (see below)	delimited string

When you use the **Mask Editor** to create a dialog box parameter for a masked block, you provide this information:

- The prompt, which you enter in the **Prompt** field.
- The variable that holds the parameter value, which you enter in the **Variable** field.
- The type of field created, which you specify by selecting a **Control type**.
- Whether the value entered in the field is to be evaluated or stored as a literal, which you specify by selecting an **Assignment** type.

The mask parameters, listed in the table on the previous page, store the values specified for the dialog box parameters in these ways:

- The **Prompt** field values for all dialog box parameters are stored in the `MaskPromptString` parameter as a string, with individual values separated by a vertical bar (|), as shown in this example:

  ```
  "Slope:|Intercept:"
  ```

- The **Variable** field values for all dialog box parameters are stored in the `MaskVariables` parameter as a string, with individual assignments separated by a semi-colon. A sequence number indicates which prompt is associated with a variable. A special character preceding the sequence number indicates the **Assignment** type: @ indicates **Evaluate**, & indicates **Literal**.

  For example, `"a=@1;b=&2;"` indicates that the value entered in the first parameter field is assigned to variable `a` and is evaluated in MATLAB before assignment, and the value entered in the second field is assigned to variable `b` and is stored as a literal, which means that its value is the string entered in the dialog box.

- The **Control type** field values for all dialog box parameters are stored in the `MaskStyleString` parameter as a string, with individual values separated by a comma. The **Popup strings** values appear after the `popup` type, as shown in this example:

  ```
  "edit,checkbox,popup(red|blue|green)"
  ```

- The parameter values are stored in the `MaskValueString` mask parameter as a string, with individual values separated by a vertical bar. The order of the values is the same as the order the parameters appear on the dialog box.

For example, these statements define values for the parameter field prompts and the values for those parameters:

```
MaskPromptString     "Slope:|Intercept:"
MaskValueString      "2|5"
```

B

Model File Format

Model File Contents

A model file is a structured ASCII file that contains keywords and parameter-value pairs that describe the model. The file describes model components in hierarchical order.

The structure of the model file is:

```
Model {
  <Model Parameter Name> <Model Parameter Value>
  ...
  BlockDefaults {
    <Block Parameter Name> <Block Parameter Value>
    ...
  }
  AnnotationDefaults {
    <Annotation Parameter Name> <Annotation Parameter Value>
    ...
  }
  System {
    <System Parameter Name> <System Parameter Value>
    ...
    Block {
      <Block Parameter Name> <Block Parameter Value>
      ...
    }
    Line {
      <Line Parameter Name> <Line Parameter Value>
      ...
      Branch {
        <Branch Parameter Name> <Branch Parameter Value>
        ...
      }
    }
    Annotation {
      <Annotation Parameter Name> <Annotation Parameter Value>
      ...
    }
  }
}
```

The model file consists of sections that describe different model components:

- The `Model` section defines model parameters.
- The `BlockDefaults` section contains default settings for blocks in the model.
- The `AnnotationDefaults` section contains default settings for annotations in the model.
- The `System` section contains parameters that describe each system (including the top-level system and each subsystem) in the model. Each `System` section contains block, line, and annotation descriptions.

All model and block parameters are described in Appendix A.

The Model Section

The `Model` section, located at the top of the model file, defines the values for model-level parameters. These parameters include the model name, the version of Simulink used to last modify the model, and simulation parameters.

The BlockDefaults Section

The `BlockDefaults` section appears after the simulation parameters and defines the default values for block parameters within this model. These values can be overridden by individual block parameters, defined in the `Block` sections.

The AnnotationDefaults Section

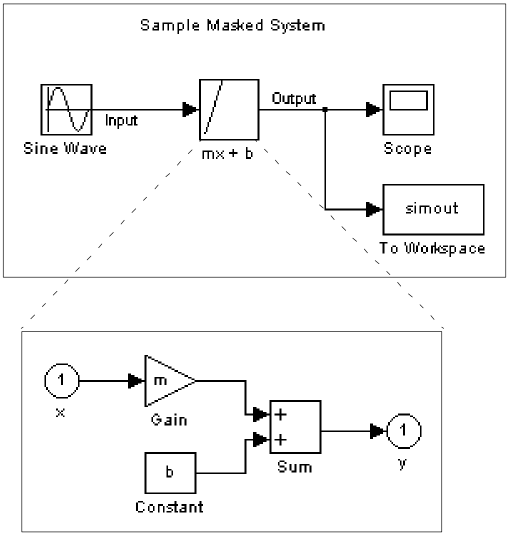
The `AnnotationDefaults` section appears after the `BlockDefaults` section. This section defines the default parameters for all annotations in the model. These parameter values cannot be modified using the `set_param` command.

The System Section

The top-level system and each subsystem in the model are described in a separate `System` section. Each `System` section defines system-level parameters and includes `Block`, `Line`, and `Annotation` sections for each block, line, and annotation in the system. Each `Line` that contains a branch point includes a `Branch` section that defines the branch line.

A Sample Model File

This model file describes the sample system that illustrates masking, described in Chapter 6. The model and subsystem look like this:



The mx + b subsystem

The model file, `mask_example.mdl`, follows.

```
Model {
  Name                   "mask_example"
  Version                2.00
  SimParamPage           Solver
  SampleTimeColors       off
  WideVectorLines        off
  PaperOrientation       landscape
  StartTime              "0.0"
  StopTime               "10.0"
  Solver                 ode45
  RelTol                 "1e-3"
  AbsTol                 "1e-6"
  Refine                 "1"
  MaxStep                "auto"
  InitialStep            "auto"
```

```
FixedStep               "auto"
MaxOrder                5
OutputOption            RefineOutputTimes
OutputTimes             "[]"
LoadExternalInput       off
ExternalInput           "[t, u]"
SaveTime                on
TimeSaveName            "tout"
SaveState               off
StateSaveName           "xout"
SaveOutput              on
OutputSaveName          "yout"
LoadInitialState        off
InitialState            "xInitial"
SaveFinalState          off
FinalStateName          "xFinal"
LimitMaxRows            off
MaxRows                 "1000"
Decimation              "1"
AlgebraicLoopMsg        warning
MinStepSizeMsg          warning
UnconnectedInputMsg     warning
UnconnectedOutputMsg    warning
UnconnectedLineMsg      warning
ConsistencyChecking     off
ZeroCross               on
BlockDefaults {
  Orientation             right
  ForegroundColor         black
  BackgroundColor         white
  DropShadow              off
  NamePlacement           normal
  FontName                "Helvetica"
  FontSize                10
  FontWeight              normal
  FontAngle               normal
  ShowName                on
}
```

```
AnnotationDefaults {
  HorizontalAlignment    center
  VerticalAlignment      middle
  ForegroundColor        black
  BackgroundColor        white
  DropShadow             off
  FontName               "Helvetica"
  FontSize               10
  FontWeight             normal
  FontAngle              normal
}
System {
  Name                   "mask_example"
  Location               [628, 63, 1128, 323]
  Open                   on
  ScreenColor            white
  Block {
    BlockType              Scope
    Name                   "Scope"
    Position               [305, 100, 335, 130]
    Floating               off
    Location               [188, 365, 512, 604]
    Open                   off
    Grid                   on
    TickLabels             on
    ZoomMode               on
    TimeRange              "auto"
    YMin                   "-5"
    YMax                   "5"
    SaveToWorkspace        off
    SaveName               "ScopeData"
    LimitMaxRows           on
    MaxRows                "5000"
    Decimation             "1"
    SampleInput            off
    SampleTime             "0"
  }
```

```
Block {
  BlockType             Sin
  Name                  "Sine Wave"
  Position              [100, 100, 130, 130]
  Amplitude             "1"
  Frequency             "1"
  Phase                 "0"
  SampleTime            "–1"
}
Block {
  BlockType             ToWorkspace
  Name                  "To Workspace"
  Position              [305, 160, 365, 190]
  VariableName          "simout"
  Buffer                "inf"
  Decimation            "1"
  SampleTime            "–1"
}
Block {
  BlockType             SubSystem
  Name                  "mx + b"
  Position              [195, 97, 230, 133]
  ShowPortLabels        on
  MaskType              "SampleMaskedBlock"
  MaskDescription       "Models the equation for a line, "
                        "y = mx + b.\nThe slope and "
                        "intercept are mask block"
                        "parameters."
  MaskHelp              "Enter the slope (m) and "
                        "intercept (b) in the dialog box "
                        "parameter fields.\nThe block "
                        "generates y for a given input, x."
  MaskPromptString      "Slope:|Intercept:"
  MaskStyleString       "edit,edit"
  MaskVariables         "m=@1;b=@2;"
  MaskDisplay           "plot( [0 1], [0 m] + (m<0) )"
  MaskIconFrame         on
  MaskIconOpaque        on
  MaskIconRotate        off
  MaskIconUnits         normalized
```

```
MaskValueString        "3|2"
System {
  Name                   "mx + b"
  Location               [561, 354, 851, 509]
  Open                   off
  ScreenColor            white
  Block {
    BlockType              Inport
    Name                   "x"
    Position               [50, 50, 70, 70]
    Port                   "1"
    PortWidth              "-1"
    SampleTime             "-1"
  }
  Block {
    BlockType              Constant
    Name                   "Constant"
    Position               [110, 103, 140, 127]
    Value                  "b"
  }
  Block {
    BlockType              Gain
    Name                   "Gain"
    Position               [110, 45, 140, 75]
    Gain                   "m"
  }
  Block {
    BlockType              Sum
    Name                   "Sum"
    Position               [185, 72, 215, 103]
    Inputs                 "++"
  }
  Block {
    BlockType              Outport
    Name                   "y"
    Position               [255, 80, 275, 100]
    Port                   "1"
    OutputWhenDisabled     held
    InitialOutput          "0"
  }
```

```
      Line {
        SrcBlock                "x"
        SrcPort                 1
        DstBlock                "Gain"
        DstPort                 1
      }
      Line {
        SrcBlock                "Sum"
        SrcPort                 1
        DstBlock                "y"
        DstPort                 1
      }
      Line {
        SrcBlock                "Constant"
        SrcPort                 1
        Points                  [25, 0]
        DstBlock                "Sum"
        DstPort                 2
      }
      Line {
        SrcBlock                "Gain"
        SrcPort                 1
        Points                  [25, 0]
        DstBlock                "Sum"
        DstPort                 1
      }
    }
  }
  Line {
    Name                    "Output"
    Labels                  "[1, 0]"
    SrcBlock                "mx + b"
    SrcPort                 1
    Points                  [35, 0]
    Branch {
      Points                  [0, 60]
      DstBlock                "To Workspace"
      DstPort                 1
    }
```

```
        Branch {
          DstBlock                "Scope"
          DstPort                 1
        }
      }
      Line {
        Name                    "Input"
        Labels                  [0, 0]
        SrcBlock                "Sine Wave"
        SrcPort                 1
        DstBlock                "mx + b"
        DstPort                 1
      }
      Annotation {
        Position                [215, 65]
        Text                    "Sample Masked System"
      }
    }
  }
```

C

The SimStruct

This table provides the names and descriptions of the macros that get and set values in the SimStruct structure, which is in `simstruc.h` located in `simulink/include`. Any macros not documented are reserved for use by The MathWorks and should not be used by S-functions. The SimStruct is the structure that describes S-functions, which are described in Chapter 8.

Macro	Description
`ssGetModelName(S)`	Get name of model
`ssGetPath(S)`	Get path of model
`ssGetParentSS(S)`	Get parent subsystem or system SimStruct
`ssGetNumContStates(S)`	Get number of continuous states
`ssSetNumContStates(S,nContStates)`	Set number of continuous states
`ssGetNumDiscStates(S)`	Get number of discrete states
`ssSetNumDiscStates(S,nDiscStates)`	Set number of discrete states
`ssGetNumTotalStates(S)`	Get total number of states (continuous and discrete)
`ssGetNumOutputs(S)`	Get number of outputs
`ssSetNumOutputs(S,nOutputs)`	Set number of outputs
`ssGetNumInputs(S)`	Get number of inputs
`ssSetNumInputs(S,nInputs)`	Set number of inputs
`ssIsDirectFeedThrough(S)`	Direct feedthrough flag
`ssSetDirectFeedThrough(S,dirFeed)`	Set direct feedthrough flag
`ssGetNumRWork(S)`	Get size of real work vector
`ssSetNumRWork(S,nRWork)`	Set size of real work vector
`ssGetNumIWork(S)`	Get size of integer work vector
`ssSetNumIWork(S,nIWork)`	Set size of integer work vector

Macro	Description
ssGetNumPWork(S)	Get size of pointer work vector
ssSetNumPWork(S,nPWork)	Set size of pointer work vector
ssGetNumSFcnParams(S)	Get number of S-function parameters
ssSetNumSFcnParams(S,nSFcnParams)	Set number of S-function parameters
ssGetSFcnParam(S,index)	Get specific S-function parameter
ssGetNumSampleTimes(S)	Get number of sample times
ssSetNumSampleTimes(S,nSampleTimes)	Set number of sample times
ssGetU(S)	Get input vector
ssGetX(S)	Get state vector
ssGetdX(S)	Get derivative vector
ssGetContStateDisabled(S)	Get continuous state for disabled subsystem
ssGetIWork(S)	Get pointer to integer work vector
ssGetIWorkValue(S,iworkIdx)	Get integer work vector element value
ssSetIWorkValue(S,iworkIdx,iworkValue)	Set integer work vector element value
ssGetRWork(S)	Get pointer to real work vector
ssGetRWorkValue(S,rworkIdx)	Get real work vector element value
ssSetRWorkValue(S,rworkIdx,rworkValue)	Set real work vector element value
ssGetPWork(S)	Set pointer to pointer work vector
ssGetPWorkValue(S,pworkIdx)	Get pointer work vector element value
ssSetPWorkValue(S,pworkIdx,pworkValue)	Set pointer work vector element value
ssGetUserData(S)	Get user defined pointer value
ssSetUserData(S,userDataPtr)	Set user defined pointer value
ssGetT(S)	Get the simulation time

Macro	Description
`ssGetTaskTime(S,sti)`	Used for multirate S-functions to get the task time of a specific sample time index
`ssGetTStart(S)`	Get simulation start time
`ssGetTFinal(S)`	Get simulation stop time
`ssIsMinorTimeStep(S)`	Return TRUE if this is a minor time step
`ssIsMajorTimeStep(S)`	Return TRUE if this is a major time step
`ssGetSimTimeStep(S)`	Get the type of the current time step, either MAJOR_TIME_STEP or MINOR_TIME_STEP
`ssSetStopRequested(S,val)`	Set to request to stop the simulation
`ssGetSolverName(S)`	Get solver name
`ssGetStepSize(S)`	Get simulation time step size
`ssSetSolverNeedsReset(S)`	Call to reset the ODE solver when the system undergoes a noncontinuous change
`ssGetSampleTime(S,sti)`	Get sample time
`ssSetSampleTime(S,sti,t)`	Set sample time
`ssGetOffsetTime(S,sti)`	Get sample time offset
`ssSetOffsetTime(S,sti,t)`	Set sample time offset
`ssSetTNext(S,tnext)`	Set the time of the next S-function call
`ssIsSampleHit(S,sti,tid)`	Return TRUE if it is a sample hit
`ssIsFirstInitCond(S)`	Return TRUE if this is the first time the `mdlInitializeConditions` is called

Index

E

M

N

O

W

X

Z